

AD-A199 271

ESD-TR-88-264

3451-4-010/2

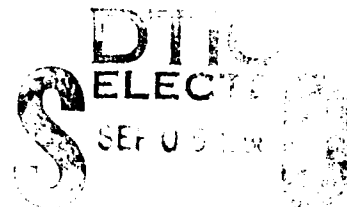
United States Air Force Program Office
Guide to Ada, Edition 4

CHRISTINE AUSNIT
ERNESTO GUERRIERI
PHILIP HOOD
NANCY INGWERSEN

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02254

31 March 1988

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED



Prepared For

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR ADVANCED DECISION SYSTEMS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

88 9 2 04 9

2


LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

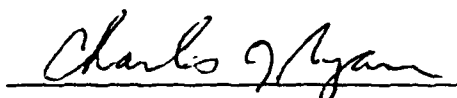
OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

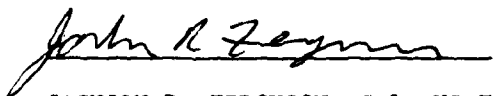


MARK V. ZIEMBA, 1Lt, USAF
Project Manager, Project 2526
Computer Resource Management
Technology Program (PE 64740F)



CHARLES J. RYAN, Maj, USAF
Program Manager, Computer Resource Management
Technology Program (PE 64740F)
Deputy Commander for Advanced Decision Systems

FOR THE COMMANDER



JACKSON R. FERGUSON, Col, USAF
Director, C³ Technology
Deputy Commander for Advanced
Decision Systems

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 3451-4-010/2			5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88- 264		
6a. NAME OF PERFORMING ORGANIZATION SofTech, Inc.		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION HQ Electronic Systems Division (AVSE)		
6c. ADDRESS (City, State, and ZIP Code) 460 Totten Pond Road Waltham, MA 02254			7b. ADDRESS (City, State, and ZIP Code) Hanscom AFB Massachusetts, 01731-5000		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Deputy for Advanced Decision Systems		8b. OFFICE SYMBOL (If applicable) ESD/AVSE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33600-87-D-0037 47337		
8c. ADDRESS (City, State, and ZIP Code) Hanscom AFB Massachusetts, 01731-5000			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO	PROJECT NO	TASK NO
					WORK UNIT ACCESSION NO
11. TITLE (Include Security Classification) United States Air Force Program Office Guide to Ada, Edition 4					
12. PERSONAL AUTHOR(S) C. Ausnit, E. Guerrieri, P. Hood, N. Ingwersen					
13a. TYPE OF REPORT Technical		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1988 March 31	
15. PAGE COUNT					
16. SUPPLEMENTARY NOTATION					
17. COSAT CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Ada, policy, simulation, benchmarks, real-time processing, distributed processing		
FIELD	GROUP	SUB-GROUP			
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The purpose of the Program Office Guide to Ada is to discuss issues affecting the selection development and maintenance of systems whose software is written in the Ada language. Each edition focuses on a different set of topics and their implications for managers. This edition focuses on Ada usage issues, policy updates, progress on benchmarks, the use of Ada in simulation, lessons learned on Ada projects, distributed processing, real-time issues, and contractor evaluation.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL M. V. Ziemba, 1Lt, USAF			22b. TELEPHONE (Include Area Code) (617) 377-2656		22c. OFFICE SYMBOL ESD/AVSE

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

EXECUTIVE SUMMARY

This report is the fourth of four Editions of the Program Office Guide to Ada. Collectively, these editions complement the Program Manager's Guide to Ada, ESD-TR-85-159, dated May 1985. This effort was sponsored by the Air Force Computer Resource Management Technology Program, Program Element 64740F, Project 2526, Software Engineering Tools and Methods. The SEI (Software Engineering Institute) has an ongoing effort to study Ada related issues and will publish handbooks on their findings on a periodic basis.

The goal of the four supplemental editions has been to discuss the transition of Ada into the life cycle phases, into different classes of software, and into the acquisition process. Software engineering concepts, methods, standards, and environments are discussed extensively. The relationship between Ada and software engineering is explored both from the viewpoint of usage and tools. Management concerns on cost, policy, and progress evaluation are also reviewed.

Edition 1 addressed: policy, run-time efficiency, run-time support environment customization, training, Ada program design languages, and conversion of non-Ada code.

Edition 2 addressed: DoD Standards 2167 and 2168, guidelines for proposal evaluation, reusability and portability considerations, software costing models, benchmarking efforts, and Ada software libraries.

Edition 3 addressed: maintenance of the Ada Standard, Ada Education and Training Study, program proving and verification, environments, tools, interfaces, and computer-aided software engineering.

Edition 4 discusses:

- Ada usage issues.
- policy updates.
- progress on benchmarks.
- the use of Ada in simulation,
- lessons learned on Ada projects.
- distributed processing.
- real-time issues, and
- contractor evaluation.

A well-written Ada program is supposed to satisfy criteria such as readability, modularity, and extendability. The proper use of language constructs and coding style enables the code to meet these criteria. Moreover, practical experience on Ada projects to date has also validated the fact that Ada "delivers" on many of the software engineering concepts it is intended to support.

Ada technology has greatly improved during the last few years. Inefficiency and tool immaturity have been the major complaints of Ada users to date. Vendors have passed the first milestone, validation, and are now concentrating on run-time system support and performance goals. Standardized sets of benchmark suites are becoming available, enabling more meaningful comparisons of Ada products. As experience with benchmarks continues to grow, guidelines are emerging on their installation and execution.

Software engineers are using Ada on a wide variety of projects, including real-time systems, distributed systems, and simulation. Ada supports both traditional real-time development and new, more flexible approaches. Some designs may require customized run-time systems, providing support for the unit of distribution, internode communication and error handling, a run-time system scheduler, and discrete event simulation. Research shows promise of a hybrid method of development that combines the advantages of the two existing methods.

In 1987 a major policy milestone was achieved with the signing of Department of Defense (DoD) Directives 3405.1 and 3405.2, mandating the use of Ada for all defense-related software. The implementation of this policy should serve as a clear signal within the DoD and industry that the time for transition to Ada is now. The fact that contractors must now use Ada has motivated studies in software productivity metrics to aid in contractor evaluation.

FOREWORD

This report is the fourth in a series of four editions that supplement the Program Manager's Guide to Ada, ESD-TR-85-159, published by The Computer Resource Management Technology Program in May, 1985. The introduction of Ada as the mandated high order language for Mission Critical Computer Programming in the Department of Defense has generated a need for clear, concise information for program managers and others concerned with cost, schedule, and performance in the application of this new language.

The intent of this series is to bring Program Office personnel up to date on facts presented in the original Program Manager's Guide, as well as to provide a more rounded discussion on certain subjects presented in the original guide. This series of four reports is designed for the program manager and his technical staff. It is recommended that the four editions comprising this report be kept with the original Program Manager's Guide to Ada, forming a ready reference to Ada and Ada-related topics.

ACKNOWLEDGEMENTS

This report is sponsored by the Air Force Computer Resource Management Technology Program, PE 64740F, Project 2526 (Software Engineering Tools and Methods), ESD/AVS, Hanscom Air Force Base, Massachusetts.

The Computer Resource Management Technology Program is the Air Force engineering development program to develop and transfer into active use the technology, tools, and techniques needed to cope with the explosive growth in Air Force systems that use computer resources. The goals of the program are to: (a) provide for the transition of computer system developments in laboratories, industry, and academia to Air Force systems; (b) develop and apply software acquisition management techniques to reduce life-cycle costs; (c) provide improved software design tools; (d) address the various problems associated with computer security; (e) develop advanced software engineering tools, techniques, and systems; (f) support the implementation of high order languages, e.g., Ada; (g) address human engineering for computer systems; and (h) develop and apply computer simulation techniques for the acquisition process.

It would be impossible to list all of the contributors, but special recognition must be given to: Capt Jack Boepple, Dr. Norman H. Cohen, Major Charles Engle, John T. Foreman, Larry Gresham, and Marlene Hazle.

Contents

Executive Summary	iii
Foreword	v
Acknowledgements	vi
22 Ada Usage Issues	22-1
22.1 Ada's Support for Software Engineering	22-1
22.1.1 Modular Structure	22-2
22.1.2 Separate Compilation	22-3
22.1.3 Abstraction	22-4
22.1.4 Structured Control	22-6
22.1.5 Reusability	22-7
22.1.6 Environment specific features	22-7
22.2 Programming Style	22-8
22.2.1 Style Attributes	22-9
22.2.2 Naming Conventions	22-12
22.2.3 Tools	22-13
23 Policy Updates	23-1
23.1 DoD Directive Status	23-1
23.1.1 Directive 3405.1	23-1
23.1.2 Directive 3405.2	23-2
23.1.3 Impact of these Directives	23-3
23.2 Air Force Policies	23-4
23.2.1 AFR 800-14	23-4
23.2.2 AFR 700-9	23-4
23.3 Policy Changes since Previous Editions	23-5
24 Benchmarks	24-1
24.1 Purpose and Description	24-1

24.2 Major Existing Benchmarks	24-2
24.2.1 Performance Issues Working Group (PIWG)	24-2
24.2.2 University of Michigan	24-2
24.2.3 Whetstone	24-3
24.2.4 Dhrystone	24-3
24.3 Additional Work	24-4
24.3.1 Production Quality Ada Compiler Report	24-4
24.3.2 SEI Benchmarks for Embedded Real-Time Systems	24-5
24.3.3 Real-time, Run-time Environment Studies	24-6
24.3.4 Ada Run-Time Environment Working Group (ARTEWG)	24-6
24.3.5 Commercial Ada Users Working Group (CAUWG)	24-7
24.3.6 Armonics Benchmarks	24-7
25 Simulation and Emulation of Systems in Ada	25-1
25.1 Simulation vs. Prototypes	25-1
25.2 On-going Simulation Work	25-2
26 Lessons Learned on Ada Projects	26-1
26.1 Metrics	26-1
26.1.1 Distribution of Effort in Different Phases	26-2
26.1.2 Productivity	26-2
26.2 Language Objectives	26-3
26.2.1 Design	26-4
26.2.2 Reusability	26-5
26.2.3 Portability	26-6
26.2.4 Language Use	26-7
26.2.5 Testing	26-8
26.2.6 Maintainability	26-9
26.3 Tools and Training	26-9
26.3.1 Impact and Adequacy of Tools	26-10
26.3.2 Ada and Software Engineering Training	26-11

26.4 Management	26-11
26.4.1 Cost Estimation	26-12
26.4.2 Resources Needed	26-12
26.4.3 Receptiveness	26-12
26.4.4 Ada Experience Forums	26-13
27 Distributed Processing	27-1
27.1 Current Development Issues	27-1
27.2 Single Program Models	27-3
28 Real-Time Issues in Ada	28-1
28.1 Evaluation of Cyclic Executive Approach	28-1
28.1.1 Description	28-1
28.1.2 Strengths and Weaknesses	28-3
28.2 Evaluation of Data Driven Approaches	28-4
28.2.1 Description	28-4
28.2.2 Strengths and Weaknesses	28-5
28.3 Temporal Models	28-5
28.3.1 Processing Models	28-5
28.3.2 Transformational Techniques	28-6
28.4 Run-Time Environment Technology	28-6
29 Contractor Evaluation	29-1
29.1 Software Engineering Exercise	29-1
29.2 Ada Decision Matrix	29-2
29.3 Process Evaluation	29-3
29.3.1 Metrics	29-3
29.3.2 Contractor Capabilities	29-4
A Appendix: References	A-1
B Appendix: Bibliography	B-1

Section 22

Ada Usage Issues

To use the Ada standard to its fullest, it is important to understand the philosophy behind the development of Ada. Good usage of Ada requires disciplined methodologies¹, using Ada features towards the application of the methodologies, and good programming style. This section covers how Ada supports modern software engineering practices, without discussing Ada syntax. Guidelines for Ada programming style are also presented.

22.1 Ada's Support for Software Engineering

Ada was designed to be a general-purpose language, facilitating the development of reliable and maintainable software. It was designed for embedded computer systems, though it has been proven useful in other applications, such as data processing. The language provides compile-time detection of many coding errors and encourages modern software engineering practices to ensure reliable code. Readability is emphasized through programming conventions and proper use of its constructs.

Ada is a large and powerful language. A programmer need not use all of the features of the language in every program. It is both normal and appropriate to use just a subset of the features. To use Ada to its fullest, a programmer needs proper training and tools.

Along with discussing the features particular to Ada, their advantages and disadvantages will be pointed out. Run-time efficiency, discussed in Edition 1, Section 3.2, will not be further discussed here. This section is not meant to be a catalog of Ada constructs; only selected features are discussed. These features are organized by topic as follows:

- Modular structure,
- Separate compilation,
- Abstraction,
- Structured control,
- Reusability, and
- Environment specific features.

¹ Programming methodologies have been discussed in Edition 1, Section 5, Ada Program Design Language; Edition 2, Section 8, DoD-STD-2167 and Methodologies for Use with Ada; and Edition 2, Section 11, Reusability and Portability.

The most versatile language feature is the package, different aspects of which are illustrated throughout this section.

22.1.1 Modular Structure

In many languages including Ada, a large program will be broken down into numerous modules to make it easier to understand. Modules usually perform a specific function that can be separate from the rest of the program. If modifications need to be made, modularity makes it easier to isolate the changes. The Ada view of the world is not a strict hierarchy, though hierarchies are permitted and encouraged. It allows for different threads of control as well as a combination of a network and a layered structure. Ada provides several kinds of program units that aid modularity, namely, subprograms, tasks, packages, and generics.

Subprograms may be either procedures or functions. They are similar to subroutines in other languages. A *task* is similar to a subprogram except it provides separate parallel threads of control, often needed in real-time or concurrent processing. *Packages* are the basic unit for structuring programs. A package is usually a grouping of similar program elements that may be used by other parts of the program. For example, a package may contain procedures, functions, type declarations, exception declarations, and tasks. A *generic unit* can either be a package or a subprogram. A generic unit is a template from which a non-generic unit can be obtained.

Ada is richer in programming structure constructs than other languages, which results in a great degree of control over the program name space², more manageable parallel software development, and also a reduction in the "ripple effect" of errors. It is generally thought that better structured code leads to better quality code, because it is easier to read and maintain. The existence of the different kinds of program units makes it important to master new structuring techniques and the interrelationships of the units.

There are some similarities and also some fundamental differences between the classes of program units. The similarities are that each have a *specification* and a *body*. The specification satisfies a "need to know" on the caller's behalf. It defines the interface to the other program units. In Ada, the information needed to call another unit is intentionally isolated from the implementation of that unit. The purpose is to minimize "coupling" and increase the independence of the units. The theory behind it is to force the programmer to think in terms of the calling interfaces. The body contains the implementation.

²Program name space refers to the set of names of program entities (variables, procedures, exception, etc.) which are accessible (can be named) from a particular point in the program execution. Modern software engineering practices encourage restricting the name space in order to isolate the effect of program changes.

Packages are a relatively new concept; from a structure point of view, packages permit an easy way to split a program into smaller, understandable units. Packages promote maintainability by localizing changes, restricting the impact of a change. This makes maintenance easier because as long as the interface (the package specification) does not change, no other programs are affected by a modification to the corresponding package body.

The ability to import a package is distinct from the capability to include a file containing data and other routines. In other languages, **include files** are generally just textual expansions at the specified location. In Ada however, packages provide a much more selective mechanism because you bring in only the specification, although the body of course will be automatically brought in so that the executable code is available at run-time. Unlike traditional **include files**, packages provide the user with a very fine degree of control over what data/functions are known and callable; mainly because a package can be imported deep in the hierarchy of the program, so that higher-level modules are unaware of it. It can be difficult to understand the static nature of packages. They are containers rather than executable units, yet the compiler generates code for them. Unlike traditional units, packages are not directly callable entities, though by importing them one can call those entities defined in the package specification.

One of the hard things to learn about designing in Ada is how to allocate declarations and code into different packages. Because packages are such a basic, fundamental unit, the software allocation to different packages is effectively the program design. The software should be allocated to minimize dependencies between packages, partly to limit recompilation when modifying package specifications.

22.1.2 Separate Compilation

Generally, separate component compilation in other languages is best described as being independent from other component compilations because there is no cross checking and external references are generated for entities not self-contained. However, Ada with a program library, provides separate compilation where each component's external references are checked against the component containing the references in the library. This separate compilation is further enhanced by having the specification and body of a component as individually compilable library units. The implication is that after the high-level design establishes some top-level packages, the software development of the modules can then proceed largely in parallel. Separately compilable units (of which library units are a subset) include package specifications, packages bodies, procedures, functions, generic units, and task bodies. Ada includes a mechanism to break out smaller, separately compilable units from a larger enclosing one.

The specifications can be compiled separately to check the validity of the interfaces.

As a result, Ada identifies errors during the design, which is less costly than discovering and correcting errors in the integration or testing phase.

The compilation of any separately compiled unit may depend on multiple program units. The significance of this lies with the ability to control the name space. Rather than import (include) a program unit at the top level of the hierarchy, it can be imported such that it is only known to a low-level subunit.

Other languages, such as C, provide independent compilation of modules which are compilable in any order. Independent compilation produces external references, without performing the static (compile-time) interface checking that separate compilation in Ada does. Because of the static interface checking between the component being compiled and the components in the library, the programmer must pay attention to the order in which he compiles the modules, although this is enforced by the compiler and/or linker. The static interface checking forces the programmer to validate the design specifications before proceeding with the package body.

22.1.3 Abstraction

Ada allows for the abstraction of data and algorithms. Features permit both data and process encapsulation.

Ada is a strongly typed language. All objects must be declared and have a specific type. As in other languages, there are predefined types. Ada also allows the user to define new types. User-defined types can have specific ranges and specified accuracy values. Being able to define the range of a type allows the user to specify design constraints. For example, a body temperature thermometer could be represented by a real number type from 85.0 to 110.0 with increments of 0.1. An advantage of strong typing is that some errors in passing parameters and in performing comparisons and computations are identified at compile time, rather than during execution. One of the potential pitfalls of strong typing is for a user to define too many numeric types and then need to perform type conversions to be able to perform operations on values in the different types. This can be avoided by creating a few basic numeric types and then using these types as the base for user-defined subtypes, enabling direct comparisons. A good up front design could avoid these problems.

Understanding types in Ada may be difficult for programmers coming from a FORTRAN or assembler background. It is likely that there will be confusion between types and objects in the beginning. Furthermore, programmers may have trouble with the concept of naming a value, as used in declaring enumeration types.

Packages allow the user to define abstract data types, one of the foundations of object-oriented programming. All of the data in the specification will be visible to a program that uses the package. A package specification can include the type declarations

for the variables used in the package, the names and parameters of the procedures and functions contained in the package, the exceptions that are defined in the package, other packages and tasks.

Besides controlling the scope by grouping all of the declarations together in a package specification, the programmer can extend the level of abstraction by "hiding" the definition of a type from a calling program by declaring it a *private type*. This class of declaration imposes restrictions on the use of the type. Although a programmer wanting to use the package can "see" the private type declaration, his code cannot "use" the information in this declaration. This is advantageous because if the calling program doesn't know how the type is defined and cannot see how the related subprograms are implemented, then the code cannot be based on an implementation detail. This is an advantage because, later, if the implementation in the package is changed, the calling program will still be valid³.

Packages support an abstraction by grouping related subprograms. The package body has the full code for procedure, function bodies and other information that the caller of the package entities does not need to know about. Often packages contain all the possible operations for a specific type. The users of the package do not know how the package is implemented⁴, so they can not make decisions based on the implementation.

The Ada language permits the *overloading* of subprograms. Overloading is when two or more distinct subprograms are identified by the same name⁵. This means that the naming of routines is much more flexible and, hence, procedure and function names are often more easily understood. Ada also allows the overloading of enumeration literals. Overloading is a new name for a concept that has existed to a degree in other languages. However, overloads in other languages have been system defined, rather than user-defined. I/O and numeric facilities have always had the same name or symbol, regardless of the parameter type. An example is $A+B$. If the parameters A and B are numeric then addition is performed. However if A and B are strings then concatenation is performed.

Overloading is useful when a user has subprograms, either functions or procedures,

³The calling program may need to be recompiled (a disadvantage), but it will not need to be modified (a greater advantage).

⁴Users here refer to the scope of the program unit importing the package. The programmer can, in all likelihood, print out the source file for the package body. The point is that he may only write code which depends on the package specification and which can reference entities named in the specification but not entities declared in the body. This is a source of confusion for Ada novices because they will find subprogram declarations both in the specification and the body. More subprograms are often declared in the body than in the specification for reasons of modularity and readability: these "hidden" subprograms encapsulate parts of the algorithm needed to implement the subprograms declared in the specification.

⁵It should be noted that package names cannot be overloaded. Variables in two different packages may, however, have the same name because they exist in different name spaces.

that perform similar actions but on different types. For instance, it is common to have a sort routine for several different types, such as integers and floating point. Overloading permits both of the routines to be called "Sort", instead of Sort.Int and Sort.Float. However there are limits to overloading, the compiler must be able to determine which subprogram to use.

Predefined operators in Ada can also be overloaded. This is good when the user defines a new type and wants to use the commonly recognized operators, such as addition and multiplication. This provides a very powerful facility for common numeric constructs based on matrices and vectors. A function which overloads a predefined operator may be called using either infix notation or the regular function call notation. Being able to use overloaded operators with infix notation increases readability.

22.1.4 Structured Control

Ada provides statements to handle program flow control, real-time actions, and exceptions. There are many statements in Ada that are similar to statements in other languages. Many of them have stricter rules associated with them to increase program logic control and increase compile-time detection of errors.

Consider, for example, the **case** statement. Every possible value of an object in a case statement must be provided an option or a semantic error will be flagged during compilation. The **others** clause can be used to cover any value that is not explicitly provided an option.

The **goto** statement is allowed in Ada, although use of it is strongly discouraged. Inappropriate use of a **goto** in Ada, will raise errors upon compilation.

Tasks permit a user to execute programs simultaneously. Tasks provide a user with parallel threads of control. With multiprocessors they may provide actual concurrency, and apparent concurrency with a single processor. The tasks can communicate with each other, and wait for each other when necessary. Through task rendezvous, Ada provides a mechanism for synchronization and data transmission.

Ada provides ways to resolve unexpected situations, which increases the reliability of the system. In Ada these exceptional occurrences are referred to as *exceptions*. Exceptions stop the sequential execution and pass the control to a different location in the program. Exceptions can be used as a mechanism to support a fault-tolerant system. Statements to rectify an unexpected situation are located at the end of program blocks. These statements are called *exception handlers*. Exception handlers can solve the problem, pass the exception to a higher level (i.e., the calling routine) or do both. If an exception remains unresolved, is propagated to the main program and is not resolved in the main program, then execution of the program is terminated. Within exception handlers, messages can be displayed, errors can be resolved, and files can be closed to permit

clean exiting of the program. There are several predefined exceptions. For example, one would be raised upon an attempt to perform an operation with a zero divisor. Users can also define and raise their own exceptions. One of the advantages of user-defined exceptions is to give the programmer control over the granularity of exceptions.

In a very loose fashion, the exception facility is analogous to a very structured **goto** to the extent that when an exception is raised, control flow must be transferred to the exception handler at the end of the current program block or to a higher level. Moreover at the end of the handler, control must be transferred immediately outside the block of the handler. In general, predefined exceptions should not be raised explicitly by a programmer because there is no guarantee that the exception was caused by the programmer-specified condition rather than some other unforeseen run-time violation.

In developing packages, the designer should think carefully about what error conditions might occur (such as stack overflow, validation failure) and whether or not he should name these conditions (i.e., declare exceptions) to allow programs using this package to trap for this exception and choose their most appropriate course of action. A detailed discussion of exception usage is found in ADA85.

22.1.5 Reusability

Usually reusable components are self contained units, that are easily transferable to other programs. Packages are good for grouping related subprograms. Often a package will contain all of the subprograms for a user-defined type.

Another aspect of Ada that supports reusability is a generic unit. A *generic unit* is a template of either a package or subprogram, from which a non-generic unit can be obtained. They are best used for programs units which replicate a single algorithm for several different types of data. An example for a generic package would be a sort routine. The same basic program could be used for integers and reals. So to use the routine, a copy of the generic unit is generated for the specific type to be used with the program. Generics need not be found only at the utility routine level. The top-level design for a subsystem could be written as a set of generic packages.

22.1.6 Environment specific features

Motivated by the needs of the embedded systems community, the Ada features **machine code insertions** and **pragma interface**, provide ways to interface with other high order languages, assembly language, machine-specific language, and the underlying hardware. Interfacing with other software, such as databases, is discussed in Edition 3, Section 20.

In Ada, the compiler determines where the code will be stored, how types and

objects will be represented and stored, etc. Representation clauses and representation pragmas permit the user to vary this. Representation clauses work with four types of clauses: length clauses, enumeration clauses, record representation clauses, and address clauses.

The *length clause* is used to specify the amount of memory the compiler should use in representing objects of a type. The *enumeration representation clause* permits the user to specify the internal codes for an enumeration type literal. The internal codes specified do not have to be consecutive, though gaps can cause operations like finding the succeeding value to be less efficient. The *record representation clause* allows the user to specify the bit-by-bit layout of each component of a record to be specified. The *address clause* permits the user to specify the storage address of an object, subprogram, package, or task. It can also associate a hardware interrupt with a task entry. This clause is useful in embedded applications where the memory location of called subprograms must be known.

The interaction between the representation specifications and the use of other language features can create some difficult cases for the compiler that can potentially result in inefficient code. Good practice would be to isolate the representation specification items in a package body, and provide an interface (i.e., the bit handling operations) through the package specification.

22.2 Programming Style

Unlike COBOL and some other languages, Ada is free format. There are no line numbers, and statements may start in any column. Therefore the programming style for Ada is left up to the user. DoD-STD-2167A has an appendix for programming style that contractors should default to, or use as the basis for defining their own.

The most important point to make about programming style for Ada is that the programming style chosen should be consistent. Among other goals, the Ada language was designed to be reusable and maintainable, both of which require that a person unfamiliar with a program be able to pick it up and understand what is being done.

A project must establish coding standards and styles right from the beginning. It is especially important to establish the standards early in the life cycle if the design is being done in Ada. Written standards allow Quality Assurance to have an explicit domain to check.

Programming style is characterized by the format of the code and the naming conventions used. Both aspects are discussed in the following subsections, as well as the types of tools which support good programming style.

22.2.1 Style Attributes

Programmers accustomed to other languages might have problems adjusting to Ada's free format aspect, i.e., there is unlearning and relearning that needs to be done. Considering that Ada is basically free format, some programmers may complain about having to follow certain conventions, but it is important because style has a direct relation to readability. Also with the increasing availability of syntax-directed editors and other tools, there is no reason for not having properly and consistently styled code.

There are some generally accepted programming conventions. These are:

- Provide a structured header comment,
- Indent nested structures,
- Differentiate reserved words from program variables with the use of upper and lower case letters,
- Use whitespace (blank lines) to enhance readability,
- Use variable names that are self-explanatory, and
- Use comments to increase understandability only where necessary.

Figures 1 and 2 show an example program which follows the general conventions listed above, augmented by the following:

- Reserved words appear in lower case letters,
- The first letter of each non-reserved word is capitalized, except for prepositions,
- "IO" in IO package names is capitalized. In general, abbreviations are capitalized,
- In a subprogram structure, the following words are aligned:
 - with,
 - procedure (or function),
 - begin,
 - end,
- The subprogram name is repeated after the word "end" and is included as a comment after the word "begin", and

```

--// AUTHOR: Jane Doe           February 12, 1986
--//
--// PURPOSE: Insert a name and corresponding telephone number in
--//           the directory.
--//
--// EXTERNAL PROGRAM REFERENCES:
--//
--// Find -- procedure which locates an entry in the directory.
--// Save_Directory -- procedure which stores directory in a file.
--// Directory -- global data structure of directory information
--//           records.
--// Maximum_Size -- maximum number of records that can be stored
--//           in the directory.
--// Current_Size -- current number of records contained in the
--//           directory.
--// Name_Type -- type for representing the name entry in directory
--//           records.
--// Telephone_Number_Type -- type for representing the telephone
--//           number entry in the directory records.
--// Index_Type -- subtype for representing indices into the directory.
--//
--// ENCLOSED UNITS:
--//
--// None.

separate(Directory_Manager)
procedure Insert (Name           : in      Name_Type;
                  Telephone_Number : in      Telephone_Number_Type;
                  Duplicate_Present : in out Boolean;
                  Space_Available : in out Boolean) is

    Entry_Location : Index_Type;

begin -- Insert

    -- check to see if room exists for additional entry
    Space_Available := Current_Size < Maximum_Size;

    Find (Name, Entry_Location, Duplicate_Present);

    if not Duplicate_Present then

        if Space_Available then -- add entry

            Current_Length := Current_Length + 1;
            Directory(Current_Length).Name := Name;
            Directory(Current_Length).Telephone_Number := Telephone_Number;

        end if;

    end if;

    Save_Directory;

end Insert;

```

Figure 1: Example of coding style

```

--// AUTHOR: Jane Doe           February 12, 1986
--//
--// PURPOSE: Insert a name and corresponding telephone number in
--//           the directory.
--//
--// EXTERNAL PROGRAM REFERENCES:
--//   FIND
--//     procedure which locates an entry in the directory.
--//   SAVE_DIRECTORY
--//     procedure which stores directory in a file.
--//   DIRECTORY
--//     global data structure of directory information records.
--//   MAXIMUM_SIZE
--//     maximum number of records that can be stored in the directory.
--//   CURRENT_SIZE
--//     current number of records contained in the directory.
--//   NAME_TYPE
--//     type for representing the name entry in directory records.
--//   TELEPHONE_NUMBER_TYPE
--//     type for representing the telephone number entry in the
--//     directory records.
--//   INDEX_TYPE
--//     subtype for representing indices into the directory.
--//
--// ENCLOSED UNITS:
--//   None.

separate(DIRECTORY_MANAGER)
procedure INSERT
  (NAME           : in    NAME_TYPE;
   TELEPHONE_NUMBER : in    TELEPHONE_NUMBER_TYPE;
   DUPLICATE_PRESENT : in out BOOLEAN;
   SPACE_AVAILABLE : in out BOOLEAN
  ) is

  ENTRY_LOCATION : INDEX_TYPE;

begin -- INSERT

  -- check to see if room exists for additional entry
  SPACE_AVAILABLE := CURRENT_SIZE < MAXIMUM_SIZE;

  FIND (NAME,
        ENTRY_LOCATION,
        DUPLICATE_PRESENT
       );

  if not DUPLICATE_PRESENT
  then

    if SPACE_AVAILABLE
    then -- add entry

      CURRENT_LENGTH := CURRENT_LENGTH + 1;
      DIRECTORY(CURRENT_LENGTH).NAME := NAME;
      DIRECTORY(CURRENT_LENGTH).TELEPHONE_NUMBER := TELEPHONE_NUMBER;

    end if;

  end if;

  SAVE_DIRECTORY;

end INSERT;

```

Figure 2: Example of alternate coding style

- If all elements of a statement cannot fit well on one line, they are continued on the next line and are aligned so that they are easily readable (frequently parameters in a subprogram declaration will each be listed on a separate line, with the colons aligned).

Other programmers follow a variation of these guidelines, depending on personal and project preference. Some of the frequent differences are in:

- Capitalization of all user defined identifiers,
- Varying the degree of indentation, and
- Varying the alignment of control structures.

22.2.2 Naming Conventions

The Ada language does not set a specified length for identifiers, though most implementations limit them to the source program's line length. For identifiers that are made up of more than one word, an underscore is frequently used between the words.

Since the length of identifiers is only limited by a program's line length, abbreviating words is strongly discouraged, except for abbreviations that are universally understood or used project wide. There are two frequently used methods for abbreviating words. The methods are dropping the vowels from the word, and truncating the word. Whichever method is used it should be fairly consistent for the project. A detailed discussion of naming conventions may be found in [ADA85].

It is usually easier to decipher code if it reads like English. Effective use of naming conventions can make comments unnecessary. Some naming conventions that aid in this are:

- Objects are usually named with nouns. Boolean objects are named as if they ask a question.
- Procedures are named with verbs to denote that an action is occurring, and
- Functions are named with a noun or a conditional clause if a Boolean result is returned.

Organizations have and should set up guidelines for naming conventions. For instance, in addition to standard program layouts, one such guide prescribes [FOF87]:

- Individual suffixes to identify data type names, package names and object names, and
- The use of full names when referring to entities inside of package specifications.

22.2.3 Tools

There are many tools available through vendors to aid users and increase programmer productivity. Two types of tools designed to aid users in writing well styled code are syntax-directed editors and pretty printers.

Syntax-directed editors aid programmers by automating some of the program code entry. This can be accomplished through the use of code templates and/or keyboard macros. Syntax-directed editors usually fill in the semi-colons and various parts of Ada constructs. The editors embody the syntax rules for Ada. The user can spend more energy writing the code and less on the proper syntax. Some of these tools also provide a way to pre-process the code into a structured format and to check for errors.

Several programming environments provide graphical ways to design the system and have tools that automatically generate program code. (See Edition 2, Section 8.4 for a discussion of some specific methods and tools.) Some systems generate frames (i.e., the general layout of the programs), while other systems produce compilable code. Often the user will then use an editor, such as a syntax-directed editor, to complete what has been generated from the graphical design by filling in type declarations and other details. Some of these tools also tie directly into document generators, so that there is a way to maintain consistency between specifications and code.

Pretty Printers format source code so that it is more readable. They ensure the uniformity of code style and format. Some pretty printers will accept user-defined format specifications.

Tools are being developed rapidly and it would be difficult to provide a complete list of all vendor offerings. Vendors exhibit frequently at the major Ada conferences, such as: SIGAda, AdaJUG, and Ada Expo.

Section 23

Policy Updates

The signing of Department of Defense (DoD) Directives 3405.1 and 3405.2 mandating the use of Ada marks a policy milestone. All the services must now write an implementation plan consistent with these directives, modifying or creating regulations as needed in order to be compliant with the Ada mandate. The Air Force regulations addressing computer resources are undergoing revision to be compliant with these directives.

The Ada Joint Program Office (AJPO) remains the focal point for Ada technology. The AJPO is responsible for the validation procedures, as discussed in Edition 1, Section 3.1. Validated compilers will now have a special certification stamp.

23.1 DoD Directive Status

Two directives requiring the use of Ada in all DoD software were signed by Deputy Secretary of Defense William H. Taft, IV. Directive 3405.1, dated 2 April 1987, discusses computer programming language policy for the development and support of all DoD software. Directive 3405.2, dated 30 March, 1987, addresses the use of Ada on all mission critical software.

23.1.1 Directive 3405.1

Directive 3405.1 states the DoD policy that Ada is required both for mission critical and for all other applications. The stated goal is to have Ada become the single, common computer language for all defense software. Major software upgrades must also be done in Ada. Programs already in full-scale development may continue to use a language other than Ada through deployment and maintenance. When another approved higher order language is more cost-effective over the application's life cycle, this language may also be used in lieu of Ada.

Approved higher order languages are:

Ada	ANSI/MIL-STD-1815A-1983 (FIPS 119)
C/ATLAS	IEEE STD 716-1985
COBOL	ANSI X3.23-1985 (FIPS 21-2)
CMS-2M	NAVSEA 0967LP-598-2210-1982
CMS-2Y	NAVSEA Manual M-5049, M-5045, M-5044-1981
FORTTRAN	ANSI X3.9-1978 (FIPS 69-1)

JOVIAL (J73)	MIL-STD-1589C (USAF)
Minimal BASIC	ANSI X3.60-1978 (FIPS 68-1)
Pascal	ANSI/IEEE 770X3.97-1983 (FIPS 109)
SPL/1	SPL/1 Language Reference Manual, Intermetrics Report No. 172-1

National Bureau of Standards (NBS) Special Publication 500-117 provides further guidance in the selection of an appropriate high order language. Directive 3405.1 supersedes DoD Instruction 5000.31 ("Interim List of Approved Higher Order Programming Languages (HOL)").

Directive 3405.1 states an order of preference for software: 1) Commercial-Off-The-Shelf (COTS) packages and advanced software technology; 2) Ada-based software and tools; and 3, approved standard HOLs. The decision of which type of software to be used should be based on an analysis of the life cycle costs and the impact on competition.

Responsibilities for the implementation of the policy outlined in 3405.1 are allocated both to the Assistant Secretary of Defense (Comptroller) (ASD(C)) and to the Under Secretary of Defense (Acquisition) (USD(A)). Both the ASD(C) and USD(A) are responsible for the insertion of modern software technology in automated data processing and mission critical systems respectively. The ASD(C) should define research areas for information system needs and provide these topics to the USD(A), who is charged with establishing software technology research programs. The USD(A) is also tasked with managing the Ada program and the maintenance of the Ada language. Furthermore, the head of each DoD component is charged with developing an implementation plan to address the issues in the directive, designating a language-control agent, implementing a waiver process to resolve requests for non-approved HOLs and establishing evaluation, training, and education programs for advanced software technologies.

23.1.2 Directive 3405.2

Directive 3405.2 mandates the use of both the Ada language and an Ada-based program design language (PDL), preferably a compilable one, on all software "integral to weapon systems." In other words, all mission critical computer software must be written in Ada and compiled by a validated Ada compiler. Ada is the preferred language for hardware test languages for Unit Under Test equipment. It is also the preferred language for unmodified COTS software used by the DoD. The use of DoD-STD-2167 and DoD Handbook-281 is strongly recommended; the software engineering principles described therein are to be applied to the production of defense software.

As with Directive 3405.1, Ada is not required for programs which are already in full-scale development, unless the software is undergoing a major upgrade. Directive 3405.2 defines a major upgrade as the redesign or addition of more than one-third the

software.

The USD(A) is tasked with coordinating the implementation of this directive. The heads of DoD components must develop a comprehensive Ada implementation plan that addresses their organization's transition plan to adopt Ada, for example, training plans, regulations, etc. Furthermore, each DoD component must have both an Ada executive official (focal point) to monitor Ada programs and an Ada waiver control officer.

Directive 3405.2 designates the AJPO as the controlling agent of the MIL STD Ada (See Edition 3, Section 15). The Air Force is responsible for providing the Ada validation facility.

23.1.3 Impact of these Directives

There are several significant points to be made about the two directives. Most significant is the scope of the Ada mandate: it encompasses all DoD software, not just mission critical software. The earlier DeLauer memorandum had addressed just software integral to weapons systems. The two directives reinforce the idea that Ada is a general purpose language and that all DoD programs, including management information systems (MIS), for example, will benefit from the use of advanced software technology. The effect will be to gain much greater computer standardization than if weapon and support systems alone were required to use Ada. The fact that the directive specifies Ada for all other applications reflects the feeling on DoD's part that the technology to support Ada both exists and is maturing.

The existence of these two directives gives industry a clear signal that the DoD is committed to Ada. For some time, industry had felt such a direction was lacking and did not have the incentive to invest heavily in Ada software engineering tools, training, or research. Several recent studies - AFC87 and DSB87 - had noted the lack of a directive and strongly urged the DoD to unite behind a strong policy statement. The signing of Directives 3405.1 and 3405.2 shows that DoD is serious and that the transition to Ada will become effective.

Waiver granting authority is delegated to each DoD component. Either the USD(A) or the ASD(C), however, may request to review waivers, as appropriate. Thus any component granting too many waivers will certainly attract attention at high levels within the DoD. Moreover, waivers cannot be granted for entire programs; waivers can only be requested and issued at the system or subsystem level.

The fact that a DoD wide directive has been signed will enable DoD components to issue enforceable regulations. Unlike the 1983 policy letter and the Draft 5000.31 regulation, Directives 3405.1 and 3405.2 are official and their guidance must be followed. Their status as Directives gives them more prominence and is evidence that the DoD is committed to language standardization. The fact that both directives are effective

immediately reemphasizes this point.

23.2 Air Force Policies

Current Air Force policy requires the use of Ada or JOVIAL on all major programs. Only validated compilers may be used. This policy is set forth in Air Force Regulation (AFR) 800-14, which must undergo revision in order to be compliant with Directives 3405.1 and 3405.2. At the time of this writing, the Air Force is still working on submitting its implementation plan to the AJPO.

23.2.1 AFR 800-14

As noted above, AFR 800-14 does address Ada. It will require several changes in order to be fully compliant with Directives 3405.1 and 3405.2. The essence of these changes follows:

- the Ada requirement is effective immediately for all programs, not just major programs,
- an Ada-based PDL requirement must be added, and
- the waiver policy for both weapons systems and information systems must be defined and agreed upon among Air Force components.

Air Force Logistics Command (AFLC) and Air Force Systems Command (AFSC) jointly issued a supplement to AFR 800-14 in September 1987, to address the life cycle management of computer resources in systems. This supplement mandates the use of Ada on all new AFSC and AFLC programs, except for automatic test equipment test programs. Moreover, major programs must use an Ada-based PDL. Software development and support must comply with DoD-STD-2167.

The AFR 800-14 Supplement provides a framework for technology transition. Each Product Division (within AFSC) and Center (within AFLC) must establish a Mission Critical Computer Resource Focal Point (MCCRFP). The MCCRFPs are responsible for the distribution of information (policy and technology related) and for the tracking and initial processing of waivers. The AFR 800-14 supplement provides detailed information on the content of waiver requests and the kinds of justification material required.

23.2.2 AFR 700-9

AFR 700-9, which is currently under revision, addresses computer programming language policy. It is expected to set the standards for communication and computer

systems, to outline the responsibilities of major commands, set the policy on programming languages, list the currently approved languages, and address the use of 4th generation languages (4GL) and specialized languages. The revised AFR 700-9 will be based on Directive 3405.1.

23.3 Policy Changes since Previous Editions

The AJPO has announced that it will not renew the federal registration of the Ada trademark. In lieu of the trademark, the AJPO has adopted an Ada certification mark to show that a compiler is validated under the Ada Compiler Validation Capability (ACVC) suite. It was felt that the certification mark, which like the trademark has a legal definition in the United States Code, was a more appropriate means to protect the integrity of the Ada language. Vendors should use the certification mark, shown in Figure 3, on literature and documentation associated with a validated compiler.

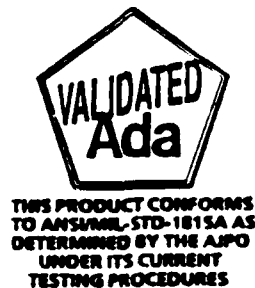


Figure 3: Ada certification mark

Section 24

Benchmarks

In earlier editions of the Program Office Guide to Ada, benchmarking was discussed with regard to performance evaluation of Ada Compilers (Edition 1, Section 6) and to the Ada Compiler Evaluation Capability (ACEC) (Edition 2, Section 13). In Section 6.3.2 of Edition 1, Execution Speed, it was indicated that comparing benchmarks for Ada to those for other languages can be quite deceiving if not performed properly. In this section, we will review some of the existing benchmarks for Ada and some of the current studies underway which will help managers select systems with the aid of benchmarks.

24.1 Purpose and Description

Benchmarks are used to perform experimentation and evaluation of various approaches in order to minimize risks to a project. In general the benchmarking process [BRAUN88] will include the following series of steps:

- Identifying critical areas.
- Analyzing alternative approaches.
- Creating and conducting experiments. and
- Applying the experimental results to the decision making process.

Benchmarks are being applied to compilers. There exist several synthetic benchmarks⁶ which are commonly used by compiler vendors to supply comparison data. Some of the more common ones are discussed in Section 24.2.

At times, it is advantageous for an organization to modify existing benchmarks or create application specific models in order to measure an Ada system's performance in a world that approximates the target environment. This might entail running applications comparable to the finished product to determine if code size and execution speed are sufficient. The Common Ada Missile Package (CAMP) Armonics (armament electronics) Benchmarks, which will be discussed in Section 24.3.6, are an example of application specific benchmarks.

⁶A synthetic benchmark is a test written specifically for the purpose of collecting benchmark data. Synthetic benchmarks are distinguished from natural benchmarks, which are usually application code adapted for the purposes of measurement.

24.2 Major Existing Benchmarks

Benchmark tests are designed to measure the capabilities of a computer system. The results are used to compare different computer systems and to determine the suitability of a computer system for particular tasks. The basic output produced by a benchmark consists of the time required to perform some task. A common technique is to write a program that performs some representative activity between calls to a system timer.

Some benchmark tests are designed to measure the performance of individual features while others combine several features in a single test. This second type, commonly known as a composite benchmark, captures the interaction of features, but it should not be construed as representative of real-time applications. In tuning a system, results of the first kind of benchmark can be extremely useful. Refer to Edition 1, Section 6.3.2, for more information on benchmark application techniques.

24.2.1 Performance Issues Working Group (PIWG)

The objective of Performance Issues Working Group⁷ (PIWG) is to provide information to the Ada community on performance issues related to Ada. PIWG has developed and distributes a test suite consisting of performance related tests.

Anyone may submit a performance issue specification to PIWG in which one requests performance measurements of special interest. The point of contact is:

Jon S. Squire
Westinghouse Electric Corporation
P.O. Box 746
M/S 1615
Baltimore, MD 21203
(301)765-3748
benchmark@ajpo.sei.cmu.edu

The PIWG suite resides in the directory ~ftp/public/piwg, on AJPO.SEI.CMU.EDU.

24.2.2 University of Michigan

The University of Michigan tests [CDV86] evaluate the execution speed of specific features of the Ada programming language. The focus is on the features from the Ada language and run-time system that are believed to be important for real-time performance. They address the issue of real-time performance measurement, with particular

⁷PIWG is an ACM SIGAda Working Group.

regard to time measurement and scheduling. This approach isolates relatively small items to allow comparisons of critical interest (e.g., rendezvous versus subroutine call). These tests have revealed some interesting observations about the effect of compiler designs: for example, certain methods used to do certain optimizations, dynamic storage and exception handling affected the benchmark results.

24.2.3 Whetstone

The Whetstone benchmark [C&W76], one of the older benchmarks, measures the performance of floating-point arithmetic. It was originally written in ALGOL 60 and subsequently translated to FORTRAN. The benchmark computations are based on the statistical distribution of statements from data collected in 1970. Programming languages have changed greatly since then, with the introduction of modern features such as record and pointer types, so Whetstone does not cover all of the features of Ada. However, the Whetstone benchmark is rich in floating point calculations, and as such is useful for comparing systems when the major concern consists of the performance of scientific calculations.

24.2.4 Dhrystone

While the Whetstone benchmark is biased towards numerical computing, the Dhrystone benchmark concentrates on systems programming. The benchmark looks at the type of data and operations performed. The Dhrystone benchmark [WEI84] used recent data on the actual use of programming languages⁸. As a consequence, the Dhrystone benchmark mainly measures the speed of non-floating point code.

The Dhrystone benchmark was developed in Ada with three guidelines: 1) the code should follow good programming practice; 2) the distribution of constructs should be weighted in favor of systems programming software; and 3) the benchmark should be designed so versions in other languages would be possible with minimal modifications. This last goal was, in fact, difficult to achieve because of the disparity of constructs in the spectrum of programming languages (e.g., FORTRAN has neither access nor task types).

⁸Programming languages used include FORTRAN, XPL, PL/1, SAL, ALGOL 68, Pascal, and Ada.

24.3 Additional Work

24.3.1 Production Quality Ada Compiler Report

The Production Quality Ada Compiler Report [HHM86], produced by The Aerospace Corporation for the Space Division of the Air Force Systems Command, is intended to provide guidelines for evaluating and selecting Ada compilers. The report defines different levels of quality compilers. It specifies the minimum requirements for a compiler, and then continues to describe the qualities of a more ideal compiler.

The evaluation of an Ada compiler to be production quality is based on satisfying the following requirements:

- Performance.
- Compiler capacity.
- User interface.
- External tools interface.
- Ada language.
- Quality assurance and reliability, and
- Documentation.

The above requirements apply to a complete compiler system (i.e., the compiler, the Ada library manager, linker/loader, and Ada run-time system). Guidelines are given to evaluate the production quality of a compiler system. The guidelines specify particular capabilities and characteristics of production quality compiler system.

Requests for copies of the report should be made to:

Aerospace Library
Reports Circulation
(M1 199)
P.O. Box 92957
Los Angeles, CA 90009
Report No. TOR-0086(6902 03)-1 or NTIS AD-A182

The report includes self-test software to enable readers of the report to evaluate the production quality characteristics of their own compilers.

An effort is underway to apply the guidelines developed in this report to commercially available compilers. Two VAX⁹ hosted compilers, from DEC¹⁰ and Telesoft, were selected. The compilers are being installed, and the suite of tests described in the report is being run. A report on the DEC results can be obtained from the point of contact listed below. The Telesoft results, as well as a revised version of the Production Quality Ada Compiler Report definition, should be available later in 1988. The point of contact is:

Lt Kurt Maschoff
Space Division
SD/ALR
P.O. Box 92960
Los Angeles, CA 90009
(213) 643-1279
(AV) 833-1279

24.3.2 SEI Benchmarks for Embedded Real-Time Systems

The Ada Embedded Systems Testbed (AEST) project at the Software Engineering Institute (SEI) is studying benchmarks to obtain data on the readiness of the Ada language and Ada tools for use in real-time systems. The focus of the project is on run-time performance, not compiler statistics. Both the PIWG and University of Michigan benchmarks are being run on VAX hosted, Motorola 68020 targeted compilers. Additional target systems under consideration include the MIL-STD-1750A processor and the U.S. Navy standard processors.

A survey of benchmarks has been conducted. [DON87]. Initial work in running the benchmarks has revealed inherent problems, documented in [ALT87] and [A&W87]. Wild swings in numbers have been reported because of differences between System.Tick¹¹ and clock resolution. Dual loop benchmarks do not necessarily remove the effect of clock imprecision. Benchmark tests need to be run for a large number of iterations in order to determine a stable result. Paging hardware, even on a bare machine, can affect timing results.

An important conclusion of [ALT87] is that Ada benchmarks are not fully transportable; in other words, the tester should expect to modify and adjust the tests for a given system. Both efficiency and performance considerations should be enumerated, for example, code generation, run-time support, tasking overhead, exception handling overhead, subprogram overhead, etc. Moreover, the underlying assumptions about the

⁹VAX is a trademark of the Digital Equipment Corporation.

¹⁰DEC is a trademark of the Digital Equipment Corporation.

¹¹The precision of the machine clock, i.e., the duration of one tick

measurement goals and their accuracy must be examined and validated in order to obtain meaningful benchmark results.

24.3.3 Real-time, Run-time Environment Studies

The purpose of the Real-time Run-time Environment Studies is to investigate the performance impact of real-time environments through the execution of benchmarks for all features of the language. The tests are implemented in Ada, JOVIAL and FORTRAN in order to allow comparison between these languages. There are also tools to facilitate comparison and analysis of test results. This effort is sponsored by the Computer Resource Management Technology Program. (Program Element 64740F), ESD/AVS, Hanscom Air Force Base, MA.

The Real-time Run-time Environment Studies is a system which contains test routines to execute language features and input/output. The focus of these studies is to create benchmarks that are very specific to individual Ada language features, in other words, to address the question, "How good is Ada relative to other languages with respect to size, speed, etc." on a feature-by-feature basis. The test suite includes some well known composite benchmarks such as Dhrystone and Whetstone.

The products of this effort are a test set and an accompanying user's manual. The test set is known as the Ada Compiler Performance Suite (ACPS). The initial version is planned for the VAX, with versions for the IBM and 1750 processor planned. The current test suite and preliminary user's manual are available from:

Rich Kayfes
Aerospace Corporation
(M1/165)
P.O. Box 92957
Los Angeles, CA 90009
(213) 336-6092

24.3.4 Ada Run-Time Environment Working Group (ARTEWG)

The Implementation Dependencies Subgroup of the Ada Run-time Environments Working Group¹² has collected information on run-time compiler dependencies. The information is being assembled into a catalog that will list the run-time implementation dependencies as they relate to the *Ada Language Reference Manual*. The catalog will give users the needed information on how the various Ada features are implemented by different compilers. It will provide insight on how applications will behave at run-time.

¹²ARTEWG is an ACM SIGAda Working Group.

See Section 28.4 for further information.

24.3.5 Commercial Ada Users Working Group (CAUWG)

The intent of the Commercial Ada Users Working Group¹³ (CAUWG) is to serve as a liaison between potential commercial Ada users, and the defense community and AJPO. CAUWG will be a focal point for the exchange of information on the Ada transitional experience. The group is investigating the barriers to the commercial use of Ada and identifying ways of resolving the technological gaps uncovered. Specifically, CAUWG is looking into Ada interface issues for:

- fourth generation languages (4GLs),
- COBOL,
- RPG2 systems,
- AUTOCODE,
- ISAM/VSAM files, and
- text processing.

CAUWG is also investigating the kind of support needed for the use of Ada in distributed processing applications.

CAUWG will document the results of its activities through guidelines on Ada products and tools, tailored for the needs of commercial users. The point of contact is:

Dave Dikel
Addamax Corporation
7799 Leesburg Pike, Suite 900,
Tysons Corner, VA 22043
(703)847-6755

24.3.6 Armonics Benchmarks

Armonics, short for armament electronics, software is one of the part categories developed under the Common Ada Missile Program (CAMP). The software has been modified to be suitable as benchmark tests for evaluating Ada system performance in this specific domain. Unlike other benchmark efforts which tend to be more language-feature or capacity oriented, these tests are application specific.

¹³CAUWG is an ACM SIGAda Working Group.

There are two series of tests: one for missile operational parts and the other for support routines. An example of support routines are the mathematical tests (e.g., trigonometric functions) which allow the user to compare characteristics such as time and accuracy across different Ada implementations. The tests are designed to measure compiler correctness as well as object code size and speed.

These harmonics benchmarks are available through the Data and Analysis Center for Software (DACS) library located at Rome Air Development Center (RADC). The DACS library functions as a distribution center; no on-line help is available. To obtain the harmonics benchmarks, requests should be sent to:

DACS
RADC/COED
Griffiss AFB, NY 13441-5700
Attn: Document Data Set Ordering
(315) 336-0937

The benchmarks are on magnetic tape medium and are available at a nominal cost. Requestors must also sign a terms and conditions statement from RADC. Several CAMP components are distributed through the DACS library: requests should specify the component clearly (i.e., CAMP parts¹⁴, harmonics benchmarks, or expert system¹⁵).

¹⁴The CAMP parts are the reusable missile software parts. See Edition 2 Section 11.2.

¹⁵The CAMP expert system was built to aid in selecting the appropriate missile parts and constructing the resultant program.

Section 25

Simulation and Emulation of Systems in Ada

Ada was not designed as a simulation language. However, the language does provide programming constructs that enable a user to build a simulation system similar to what would be possible with a simulation language.

Although, simulation languages are very diverse, they will all provide a user with simulation control, random stimuli generation, and statistics collection. Some also provide scheduling and garbage collection. Two of Ada's features that are very useful for implementing simulation systems are tasks and packages. Tasks provide a way to do concurrent processing, and the task rendezvous allows for synchronizing concurrent entities. Packages promote modularity in addition to data and process encapsulation.

There are many diverse types of computer simulation used today. They are generally grouped into four classes. Monte Carlo, continuous, discrete-event, and combined discrete-continuous, each with a special type of system that it is used to model. Ada can be used for any of these types of simulation, but some specific applications may require specialized run-time system support. A discussion of the use of Ada for discrete-event simulation can be found in [SHO87], [A&S87] and [BRU84]. The general conclusion has been that although Ada has some weak spots (in inheritance and garbage collection of allocated task bodies), it is quite well suited for this type of programming.

25.1 Simulation vs. Prototypes

Simulation and prototyping are related concepts. Their difference is mostly one of purpose. Prototypes are generally of a subset of an implementation. They are designed to show the functional capabilities of a portion of the system. On the other hand, simulations usually mimic various aspects of a system's behavior.

At times a simulation will be used with a prototype. The simulation would mimic the environment in which the prototype executes its functions. The relationship between prototypes and simulations is analogous to the one between a module and its test driver.

To decrease the level of risk on projects, the use of prototyping and incremental releases is being advocated. Critical portions of systems are prototyped or simulated early in the life cycle to determine the feasibility of the system. System design reviews often include demonstrations, either in the form of a prototype or a simulation.

25.2 On-going Simulation Work

There are an increasing number of investigations using Ada in simulations. In general Ada is being chosen for several reasons:

- High Order Languages are easier for programmers to use,
- Ada packages enhance modularity and maintainability,
- Ada provides multi-tasking,
- Ada supports dynamic allocation of memory,
- Ada has an interface to low level I/O,
- Strong type checking prevents data in low level modules from being accessed by accident,
- Record types provide a means of storing arbitrary types of data, and
- Ada supports linked list manipulation, needed for data collection and event calendar management.

Several efforts are using Ada for simulations in the systems area as well as the applications area. In [M&G87], a general purpose discrete-event simulation package is described. A queueing network simulation package is discussed in [HAS88], a motor plant simulation in [KIM88], and a spacecraft dynamics simulator in [BGA88].

To support ESD's investigation of alternative Battle Management Command, Control, and Communications (BM/C³) architectures, the MITRE Software Center is involved in simulating real-time, space-based battle management functions. Their simulating software displays three-dimensional state vectors on a workstation. They plan to support other Strategic Defense Initiative (SDI) simulations by designing the framework. The point of contact for further information is:

SDI Simulation
The MITRE Corporation
Burlington Rd
Bedford, MA 01730
(617) 271-4501

Two aircraft training simulators were redesigned and reprogrammed in Ada. The objective was to apply a modern software engineering approach, partitioning the system based on data flow and object abstraction techniques. Tasking was used sparingly in these simulations for efficiency reasons. Further information is available through:

ASD/YWB
Wright Patterson AFB, OH 45433
(513) 255-7177

Section 26

Lessons Learned on Ada Projects

The Ada language has been used on projects throughout the military - on pilot projects, applications under development, and deployed systems. In some cases, the use of Ada has been by choice; in others, it has been required. A baseline of experience is emerging, and the consensus is that Ada systems are, in fact, feasible. Ada development is by no means problem free, largely due to immature tools and to the need for new software engineering approaches and attitudes. There are clear benefits, however, from the use of Ada. The distribution of time in the life cycle is altered, with a greater proportion of time being spent earlier, in the design phase, with the result that the generated code is more correct.

In theory, using Ada is supposed to bring all sorts of benefits, in particular, lower cost and improved quality of source code. Practically speaking, however, managers who have not yet undergone the transition to Ada are asking, is all the propaganda really true? In reading about Ada one often just hears the two extremes: the system can be done in Ada or the system cannot be done in Ada because Ada is too big, too slow, etc. Results from the field are available, and this section will try to present the lessons taught by this early experience with Ada.

This section presents a summary of the Government's and its contractors' experience with Ada projects. The project database represents both large and small projects in terms of money and size. Different aspects of Ada experience are discussed in order to give a better perspective on what to expect in the transition to Ada.

26.1 Metrics

The ever increasing cost of and demand for software is focusing more attention than ever on software productivity and quality. Several metrics are used ranging from the the program size in lines of code per unit of time, to the frequency of software failure. Regardless of the aspect being measured, productivity is usually defined as a ratio of the outputs produced by a process to the inputs it consumes [BOE87]. According to Boehm, the greatest problem exists in defining the outputs. Partly because of the deficiencies in the traditional metric of delivered source lines, alternative units of measure based on program complexity have been advanced. Current efforts are aimed at establishing a set of baseline metrics to help project personnel monitor the quality and status of software projects [KEN87].

Productivity data on Ada projects is becoming available. Because of the newness of Ada technology and some unwillingness on the part of projects to release data, these

numbers should not be viewed in absolute terms. They are, however, indicative of some important trends that show that using Ada will have an unmistakable impact on software development and cost.

26.1.1 Distribution of Effort in Different Phases

Numerous data collection efforts reveal that Ada projects are front-loaded. Both first time and repeat Ada users have found that much more time is spent in the design phases, and much less time in the code and integration phases. The traditional distribution of software development effort across the design, code and test/integration phases is 40-20-40, whereas Ada tends to be more 50-15-35 [REI87]. Several projects have found that integration took less effort than originally planned.

There are several reasons explaining why Ada projects tend to be front-loaded. This distribution reflects greater attention being paid to software engineering, which encourages greater effort and thought in the early phases. The Ada language contains powerful program structure features which are available to the designer when Ada is also used as a PDL. Ada must be learned earlier in the life cycle than other languages, because it is also used as a PDL.

The distribution of effort in the software life cycle should also be examined from the point of view of the software standards. Users have noted that the software development standards, including DoD-STD-2167, are not wholly compatible with Ada software development. Complaints focus on the fact that they do not allow for evolutionary software development. The use of Ada as a design language increases the problem because it is difficult to differentiate between the expression of the design and actual code. Most standards require that the PDL be frozen before any code starts. The use of Ada compounds the problem because in Ada, the PDL is continually refined into code. In the course of this refinement, changes will probably need to be made to either the requirements or design, and when the design is expressed in an Ada PDL, it appears to be duplication of effort to update one set of files representing the design, and an almost identical set of files representing the design as partly expanded into code.

26.1.2 Productivity

A traditional measure of productivity is in delivered source lines. In earlier languages, for instance FORTRAN, a line held a whole statement (or most of it). In Ada, free formatting is encouraged for readability, and a line taken literally may well be blank or contain only a small part of a statement. Strictly speaking, there will probably be many more lines (in the sense of FORTRAN card images) of code in an Ada program than in a program in some other language. The separable characteristics of Ada speci-

fications and code provide an additional quantity of semicolons to reflect specifications and context clauses. These increase the semicolon count in Ada necessary to reflect an equivalent structure in some other language. In some cases this could be a 26% overhead in Ada and quickly offset a perceived productivity increase. Furthermore, additional Ada language constructs are used to define context and interfaces and these add to the semicolon count and should be considered when comparing an Ada effort against another language such as FORTRAN. Consequently, a revised measure is emerging, based on counting semicolons. Various considerations apply in counting semicolons, as noted in REI87.

Several attempts have been made to calculate Ada productivity in terms of semicolons per person-month. The average for these estimates is in the range of 277 to 310 H&S87 and REI87 semicolons per person-month, higher than the industry average of 200 lines of code per person-month. Another study reports a 23% increase in productivity over the life of an Ada project FOF87, consistent with the increase reported in the Harbaugh and Reifer studies¹⁶. Reifer acknowledges that there is a fairly wide range of productivity numbers collected in his research, reflecting the steep learning curve for Ada. The important lesson here is the trend to higher productivity.

The higher productivity figures must be interpreted in light of the steep learning curve for Ada. Numerous sources have found that the productivity gains will not be found in the first two or three Ada projects and, in fact, the first couple of projects may show a productivity decrease. Ada requires a heavy up-front investment in training, tools, and experience base. The theme of the December 1987 SIGAda conference was Ada usage, and many of the articles in the Proceedings, as well as studies published by the Armed Forces Communications and Electronics Association (AFCEA) and Defense Science Board [AFC87 and DSB87], point out that this investment is part of the cost of the transition to Ada: an investment that must be made in order to realize the benefits promised by the use of the Ada language.

26.2 Language Objectives

The purpose of using Ada is not just for the language but also to accrue Ada's long range benefits to the entire software engineering effort. In looking at Ada experiences, we have to consider not only the programmers' experience with individual language features, but also issues like design, reusability, portability, language use, testing, and maintainability.

¹⁶Harbaugh's data covers one project, the Graphic Kernel System (GKS) Ada implementation, while Reifer's database includes 41 projects.

26.2.1 Design

Lessons learned in the design phase fall into two major categories: the use of a design method derived from object-oriented principles and the use of Ada as a design language. In general, an object-oriented approach refers to a data-driven design philosophy. Software is viewed in terms of objects and operations on these objects, rather than along strictly functional lines.

The danger with some of the object-oriented approaches is their tendency to oversimplify the design process. Criticisms of various methods include [N&S87]:

- Lack of guidance on designing concurrent processes.
- Too deeply nested hierarchical structure.
- Limited domain of applicability to small, well-defined and well-known problems, and
- Awkward and impractical strategy, reflecting more art than engineering discipline.

Object-oriented design has become popular in the industry and its precise meaning differs among different people. In choosing this approach, or any other methodology, caution should be exercised to make sure that the design technique provides not only a disciplined decomposition of the internal functions but also guidance on "the packaging of software modules into Ada tasks, packages, and subprograms [N&S87]."

Ada has been found to be a good language to use for expressing design. Some projects have, in fact, used Ada as a design language and another language as an implementation language.¹⁷ The primary reason lies in the program structure facilities that exist in the language, in particular the package feature. There is disagreement within the Ada community as to whether full Ada or restricted Ada should be used when recording a design. The restrictions are aimed at preventing the designer from coding prematurely, for example by disallowing statements within the PDL. Many Ada program design languages (PDLs) define classes of structured comments in order to allow the designer to express other information which is not directly expressible in Ada, such as timing constraints and input assumptions. (Edition 1, Section 5 discusses PDLs.)

In spite of this drawback to using Ada as a PDL, namely the risk of premature coding, there are major benefits which have been realized. Assuming that the PDL can be machine processed (often by an Ada compiler), syntactic and semantic errors are caught much earlier in the software life cycle, in particular interface errors, such as one module calling another one with the wrong name or the wrong types of parameters. Because the PDL and the coding language are the same, creating the PDL effectively

¹⁷Directive 3405.2 states that an Ada-based program design language will be used for software design.

creates the code skeleton. Furthermore, because the code is completed through successive iterations of refining and expanding the PDL, the software development process is more evolutionary in nature, allowing for improved error detection and correction.

The point to be made here is that a compilable PDL should be used. One project has said that they suffered because their PDL was not compilable and, therefore, certain clean interface errors were not caught. Many published articles encourage the use of a machine processable PDL, i.e., an augmented compiler which can machine process the annotations. Such an enhanced compiler would not need necessarily to generate code from the PDL.

There is no agreement in the community as to whether Ada should be used during all stages in the life cycle. Some analysts feel that Ada was not designed to express system structure or system constraints. They feel that graphics or some other annotations are needed to capture temporal information and other constraints. A very persuasive argument against the use of Ada too early in the life cycle, such as during requirements definition, has been advanced: Ada is part of the solution, and requirements definition involves determining what the problem is. It is not a good idea to explain a problem by describing its solution. Also, the solution process must be defined before the solution technology can be selected. Ada is a tool, not a process, and it is the process which implies the tools, not the other way around.

Neither Ada PDL nor object-oriented methods have been found to be sufficient on their own. Used together they can be a powerful combination, in particular when the object-oriented method communicates the design through a graphical notation. The use of integrated tools supporting both the graphics and PDL is extremely important.

26.2.2 Reusability

One of the promised and highly promoted benefits of using Ada is reusable code. Ada experience to date shows that reusability does not happen by accident. The design of the software must have reuse in mind, and management must plan for reusable modules as a product. Reusability is not free; in fact, one study shows it to be a major factor influencing cost and size estimation REI87.

In discussing reusability, one should identify the level of reuse: reuse of utilities versus subsystems, and reuse within one organization or within the superstructure of the organization. Reuse has become a catch phrase, and sometimes it may only refer to utility libraries. Whenever a programmer writes a generic and instantiates it, he is "reusing" code.

Reuse costs time and money. It must be designed into the system and it affects design decisions. Edition 2, Section 11.2 addresses issues of motivation, design, and incentives. In designing reusable components, it is imperative to do a thorough functional

domain analysis. This analysis should be done as a cooperative effort between domain expert(s) and software engineers. Management must plan for both good communication and an effective retrieval system. If other projects do not know about the reusable components and cannot access them easily, reuse will not occur.

A key element in planning for reusability is the creation of an effective library and configuration management system. The Ada program model consists of a single library. Different Ada projects, each with their own library, cannot share modules easily unless the source code is duplicated for each project. Should these reusable units be changed, the changes must be both propagated to each reuser and recompiled within each project library. Tools are needed to facilitate the concept of multiple libraries [DAU87].

Several reusable libraries exist, in the commercial and public domains. (See Edition 2 Section 14.) In addition, there are two major government funded efforts to create reusable components, the Common Ada Missile Package (CAMP) for missile components (Air Force Armament Lab) and the Reusable Ada Packages for Information System Development (RAPID) for a library/retrieval system (Army Information Systems Engineering Command). The Software Technology for Adaptable Reliable Systems (STARS) program is expected to fund a reusable software repository. It is significant that there is sufficient high-level interest to create the environments needed to support reusable components, as manifested in these three government sponsored efforts.

26.2.3 Portability

There is limited experience in porting Ada code. Few applications have been ported across different host machines. Ada tool vendors have the most experience in porting Ada code insofar as they offer products which run on different hosts. As with most languages, machine-independent portions of code are easier to port. Edition 2 Section 11.3 discusses portability in depth.

An important consideration in porting Ada is the tool set on the target machine. In some cases, the development and execution environment are different. The target environment needs an adequate tool set for testing and debugging purposes, so that productivity losses are not incurred. When the development and support environments are different, there should be a strong relation between the two, ideally a cross compiler. Without such a facility, all code needs to be recompiled after it is ported. In theory, this should not be a problem but, in practice, the compilers may not be of the same maturity and will not necessarily generate code of equivalent quality for all Ada constructs. Moreover, there are risks that the compiler on the target machine cannot handle optional Ada features related to representation specifications (Chapter 13 of the *Ada Language Reference Manual*) and that it has a different set of bugs requiring different workarounds than on the development system.

Configuration management is an issue in portability, especially in the situation where source code is being developed on several machines. Experience indicates that a great deal of automation is required to control source code and maintain stable baselines.

Benchmarks have been ported frequently and have required some editing to compile and execute successfully. Compiling benchmarks for the first time may reveal errors in the compiler or in the benchmark. The benchmark may assume pragmas that have not been implemented on a given system. In running benchmarks on embedded computer target systems, modifications may be needed in order to link to the target input/output functions. A more detailed discussion of benchmark experiences is in Section 24 of this Edition.

26.2.4 Language Use

Users' experiences with Ada language features are very positive with respect to Ada's strong typing and program structuring constructs but negative with respect to existing implementations of Ada run time systems and Ada's tasking paradigm. Ada has made certain application code more efficient, for example, slices. The use of Ada has led to much cleaner implementations of the designs, which are easier to understand and work with at integration time.

The biggest problem, however, lies in immature tool sets and the lack of efficient run-time support. Early releases of compilers did not support the machine-dependent interface of the language, causing problems for many real-time applications. Early implementations of generics resulted in too much code expansion. The tasking overhead, especially that for rendezvous and the associated context switching, is too high for most applications. A high level of CPU reserve is needed in order to run Ada software. Problems have been reported with the implementation of Ada's input/output features for some applications. Many of these problems have been resolved in subsequent releases of compilers.

Run-time systems need to be integrated with existing libraries and system services such that the application can place appropriate calls to those modules. The language-defined interface pragma needs to be supported. Run-time support libraries need additional flexibility to accommodate custom hardware chips as well as built-in test, bit manipulation, and fast interrupt capabilities.

The Ada Real-Time Environments Working Group (ARTEWG) has studied the deficiencies and requirements for Ada run-time support extensively. Their recommendations focus primarily on tasking, including standard application-specific scheduling algorithms packages, language clarifications, and guidelines on the use and implementation of task priorities. In a white paper published in the December 1987 Ada Expo Proceedings [ART87], the ARTEWG discusses shortfalls in current Ada run-time tech-

nology. The ARTEWG advocates systematically addressing these problems in order to develop Ada run-time idioms that meet embedded system space and time constraints as well as fault-tolerance, distributed computing, and multi level security requirements.

Although these inefficiencies are major, experience shows that they are not necessarily a reason not to use Ada. Workarounds have been found in almost all instances. The issue of excessive tasking overhead is being addressed by individual vendors, and there are various solutions being proposed, including possible language changes¹⁸, implementation-defined pragmas, and interfacing to assembly language where necessary. Successive releases of tool sets have been showing substantial improvements in performance and functions.

26.2.5 Testing

In several projects which gathered statistics on error rates, it was found that the use of Ada reduced errors by 25% to 30%. Significantly more errors were found before the integration phase, in several instances reducing the amount of time spent in this phase. This reduction in errors was attributed to the extensive static checking performed by the Ada compiler. By compiling their designs, users found that they could eliminate the syntactic and semantic errors between module interfaces early in the software life cycle, without the difficulties normally encountered during the integration phase.

Testing Ada programs has revealed interesting results. The use of certain Ada constructs such as exceptions, generics, tasking, and packages requires a different approach to testing. More special-purpose drivers are needed in order to conduct unit tests. The nondeterminism of task scheduling makes it more difficult to test concurrent programs because the failures may not be repeatable. Testing effectiveness has increased due to some Ada features. The information hiding supported through the proper use of packages has reduced regression problems. Strong typing and program unit specifications have automated detection of interface errors. The existence of exception handling has encouraged programmers to think about potential exceptional situations, thereby reducing errors.

Since the initial release, the validation suite has become more thorough in its coverage of language features. A validated compiler, however, is by no means a guarantee of a perfect, error-free compiler. The validation suite tests conformance to the standard; it provides no guarantee of performance or adequate run-time support. Over time, however, with successive releases of the validation suite, some of the harder-to-test features are being covered. For example, with release 1.9, there are tests on tasking, *Ada*

¹⁸The International Real-Time Ada Issues Workshop identified several areas where language changes would help make Ada more responsive to the real-time embedded community, such as fault-tolerant non-distributed execution, program reconfiguration, and hardware interrupt priorities [WOR87].

Language Reference Manual Chapter 13 features, and fixed-point types. As discussed in Edition 1, Section 3.1, however, the validation suite can never be complete and test every nuance of Ada.

26.2.6 Maintainability

Little data is available yet on the cost of maintaining systems written in Ada. Based on the fact that integration is easier because the Ada constructs are more understandable, maintenance should also benefit from Ada-coded programs. Software managers expect that because Ada allows for a better representation or expression of software structure and function, maintenance should be easier.

The bigger issues in maintenance may well be the need for recompilation and for configuration management. Depending on the interconnections between modules and their location in the program dependency tree, large amounts of the program may need to be recompiled. In other languages, such massive recompilation would be more characteristic of a new release of a compiler than of a change to an existing module. At first glance, this may seem to be a substantial drawback to the use of Ada. To the contrary, this forced recompilation will catch many other errors which might otherwise creep into the system, namely the notorious ripple effect errors. Sophisticated incremental recompilation tools will also help in limiting recompilation by analyzing the impact of code changes and only making those units obsolete which are affected by the change. Good design of a system may avoid most, if not all, of these problems.

The need for configuration management throughout all phases of the life cycle is important. The Ada program library model enforces a current configuration because the insertion of an updated module in the program library automatically invalidates any dependent units in the library. Because of the nature of large systems, however, more sophisticated configuration management has been found necessary in order to track baselines, versions, and variants. Moreover, such a system must track not only code but also all associated documentation (design, requirements, user manual, etc.). Data on several projects reveals that configuration management played a much more important role than anticipated.

26.3 Tools and Training

Recently, tools and training are receiving considerable attention. Ada conferences regularly have exhibit halls, which attract at least as much attention as do the regular sessions. Training was perceived as a possible cause of the slowness of the Ada transition, and the AJPO and AFCEA organized a task force to investigate the problem. The general conclusion was that training per se was not the reason projects had had problems,

but that tools were a major contributor.

26.3.1 Impact and Adequacy of Tools

As stated in Section 26.2.4, tool immaturity caused many of the problems on Ada projects. Production quality language tools are needed, such as symbolic debuggers and tasking analyzers. Tools to automate other phases of the life cycle are also needed, such as design tools to manage complexity and to help understand vertical and horizontal relationships in the software. For example, a dependency analysis tool would be useful in order to list both the units which depend on a given unit (in order to determine the modules to be recompiled should this unit change) and the units on which a given unit depend (i.e., the ones imported through **with** clauses, to facilitate debugging).

Ultimately, a fully integrated environment is needed. Users with more integrated environments have experienced much higher productivity using Ada than those with poorly integrated tools. The development environment should provide sufficient target support to allow testing and debugging.

Vendors have made great progress with Ada tools. Looking back over the history of Ada, the first goal was to develop validated compilers, ignoring considerations such as speed, object code quality, and run-time efficiency. Ada compiler development has been pushing out the limits of compiler and run-time system technology. Having met the first goal of validation, vendors are working on their next objective, performance and optimization. Successive compiler releases have made tremendous gains in speed, efficiency and correctness.

There are a great variety of host/target combinations today, showing that industry believes that there is a market and that DoD is committed to Ada. In spite of the problems that have existed with the early tools, the Ada pioneers in the user community have shown that Ada is possible. Having surmounted many obstacles in the tools area and given the variety of tool options today, they stress that the lack of a compiler for a target machine is a poor excuse. One of the recommendations coming from these users is to look for an acceptable tool set before starting the project. Developing tools at the same time as the application is a frequent source of technical, cost and budget problems. These users also stress that a pragmatic rather than a purist approach is needed. In other words, a program manager should recognize that most of a program can be done in Ada, and that it is acceptable to isolate critical portions to be done through an interface to another language. Just because the all the code cannot be written in Ada, it should not be taken to mean that none of it can be written in Ada.

26.3.2 Ada and Software Engineering Training

In both government and industry, different levels of personnel in the development, management and support functions require training in aspects of Ada and software engineering. Technical training in language features and design methods is needed for the software engineers. Managers need to learn about the impact of Ada on the life cycle and its effect on planning, scheduling, and resource allocation. Furthermore, they must recognize the need for training, both for management and non-management people. Support personnel, including acquisition, configuration management and quality assurance personnel need some familiarity with the language, the software process and software metrics.

Both government and contractor personnel agreed with the need for different kinds of training in software topics for different categories of work. Training is needed not only in writing software as an engineering discipline but also in a life cycle approach to software management and control. The rotation of uniform personnel in project management positions breeds a short term, a two to three year view of a program, rather than encouraging a long term, 20-year, life cycle attitude.

Timeliness and hands-on experience were found to be key factors in successful Ada training. Videotapes and computer-aided instruction were found to be much less effective teaching vehicles. Ada language training too far in advance of design and coding lost much of its effectiveness. In order to gain a deep understanding of Ada concepts, laboratory exercises were invaluable. The access to the compiler enabled students to overcome the hurdles of a new language's syntax and semantics and to gain confidence in the use of Ada structures. Practice in designing package specifications was characterized as an integral part of a successful technical training curriculum.

Ada training involves retraining and teaching programming attitudes. Some concepts are hard to learn for programmers with a FORTRAN or assembler background, such as those related to strong typing. Recent graduates learned the Ada language faster because of their academic experience with Pascal and other modern languages.

26.4 Management

The use of Ada is leading to changes in management expectations and planning. The transition to Ada has given new prominence to software issues such as reliability, maintainability, reusability, methodology, integrated toolsets, etc. Cost estimation and resource allocation for Ada projects are different than for non-Ada projects. The steep learning curve for Ada means that planning for the initial project will differ from planning for subsequent projects.

26.4.1 Cost Estimation

Existing cost estimation models will need to be recalibrated. Elements such as reusability, the degree of real-time processing, the learning curve, and the use of Ada program structure features need to be integrated into the models. Moreover, as the base of Ada projects grows, the weighting factors applied to the cost parameters can be better computed. Current weighting factors are no longer valid because of the different distribution of effort observed in Ada projects. During the transition period for Ada technology, prior Ada experience is an important cost factor to incorporate because of the steep learning curve characteristic of the first one or two projects.

26.4.2 Resources Needed

Management planning must account for personnel, software, and hardware resources. The initial Ada investment is expensive, involving training, tool acquisition and, in some cases, larger computers. With subsequent projects, the need for major outlays to support Ada declines substantially. The lack of experienced Ada designers and programmers on initial projects adds time to the schedule. Managers have a choice of training their own engineers or of hiring already trained personnel. In order to insert Ada technology more effectively into their own organization, managers may want to hire an "Ada guru" as a technical focal point. Where a successful transition to Ada has been achieved, there has been a broad-based management commitment to the use of Ada.

Many projects have identified a need for integrated tools to maximize the productivity gains made possible by Ada. In addition to production quality compilers, integrated design, metrics and configuration management tools are needed. Such extensive automation adds to the cost of the first few Ada projects in an organization.

26.4.3 Receptiveness

Ada has been greeted both with enthusiasm and with resistance in government and industry circles. Acceptance of Ada is gaining, with the realization that the DoD is committed (evidenced by the two Directives 3405.1 and 3405.2) and with the availability of more mature language tools. Some projects were successfully done in Ada in spite of management skepticism (either on contractor or government side). The commercial world has shown interest in Ada, and several companies have made business decisions to convert to Ada, convinced of Ada's long term, life cycle benefits. The Ada language is regularly referenced in software-related articles.

Resistance to Ada can be attributed to a combination of psychological and technical factors. Because of its newness and the lack of widespread Ada experience, project managers are afraid of failure. They do not want to take the risk on their system,

preferring to let others prove the technology. The Ada technology was not mature when the first Ada projects started, and the technical difficulties experienced were sizable. There are still technical challenges today that require innovative solutions. Ada was initially marketed as the universal problem solver for the software engineering crisis. Because this promise was not fulfilled immediately, some managers became skeptical about Ada's capabilities.

It is important to understand that the Ada language alone is not the entire solution; it is one piece in a much larger integrated solution which encompasses language, tools, and methods.

26.4.4 Ada Experience Forums

Several case studies and workshops to assess Ada experience to date have been conducted. Two companies were contracted to redesign and redevelop in Ada two aircraft training simulators. Reports on their experiences were published. An "Ada Simulator Validation Program Workshop" was held and briefing slides are available through the following address. Further information is available through:

ASD/YWB
Wright Patterson AFB, OH 45433
(513) 255-7177

The Electronic Industries Association held a workshop in November 1987 which addressed Ada experiences as well as industry questions on DoD-STD-2167 and DoD-STD-2168. The Ada Information Clearinghouse is continuing to accumulate a database of DoD programs currently using or planning to use Ada. They have developed a survey form to collect data on:

- program name and sponsor,
- point of contact,
- brief functional description of software.
- host and target systems,
- tools (compilers, design tools, etc.)
- estimated size,
- productivity,
- education and training,

- life cycle costs, and
- lessons learned.

The Ada Information Clearinghouse may be reached at:

Ada Information Clearinghouse
3D139 (1211 Fern St., C-107)
The Pentagon
Washington D.C., 20301-3081
Attn: Ada Usage

Section 27

Distributed Processing

A distributed system is a collection of processing nodes connected by some limited bandwidth communication medium. Each node is composed of one or more processing elements sharing a single memory space. Communication between nodes is usually done by message passing.

Most current applications are composed of a few distinct programs that are distributed to different nodes. These programs communicate via a message passing protocol. There are new machines and new applications being developed for which this approach is impractical due to size or the number of processing elements. This section discusses both types of projects and their relevant issues.

Historically, there has been another class of systems called "tightly coupled distributed systems." These systems are composed of several processors that communicate by means of shared memory. Today these systems are usually referred to as multiprocessors. Ada is generally considered to support this type of system very well, and most of these systems have an Ada capability.

27.1 Current Development Issues

The typical distributed application today runs on a few nodes and is developed as one or more separate programs per node. Each node in these systems is either a single processor, or a set of processors that share a common memory address space. Ada is well suited for developing software on one node, and the details of internode communication can be easily hidden in one or two packages (R&Z83 and DZM87).

There are several areas of interest for this type of distributed application in Ada:

- local area network interface,
- configuration management,
- distributed data management,
- migration of data,
- migration of code, and
- load balancing.

The most popular communication system for distributed systems is the local area network (LAN). These networks provide relatively high bandwidth communications between nodes. The communication software for most of these networks was not designed for Ada, and some of the Ada interfaces are difficult to work with. There are no inherent problems in creating a good interface, but a program manager should plan for solving a few problems when interfacing to an existing product.

Configuration management is difficult when many different programs must interact in a consistent fashion. Spreading programs across many nodes makes this problem even more difficult. The user must maintain strict control over the version of each program loaded on each node to ensure correct system operation. Components such as communications packages that are used by several programs can further aggravate this problem. Configuration management is a major factor motivating the single program approach.

There are two reasons for data to move between nodes, performance and survivability. Data can be moved to another node to balance the load across the system, as discussed later. Data may also be replicated on many nodes to provide local data access (which is faster) and to ensure the data survives the loss of a node. Replicated data, however, may require very expensive support [DDM87].

Migration of code allows a node to do the processing originally allotted to another node. Allowing multiple nodes to perform any particular processing allows a system to survive the loss of a node or to balance the load between nodes, thereby increasing overall throughput. Several issues should be examined if this mechanism is to be considered:

- How long does it take to move the code from one node to another.
- How much does this cost, and
- How are individual processor capabilities matched with the requirements for each unit of work?

It is often less expensive to have an idle copy of the software waiting on all possible host processors, triggering them as needed.

Load balancing involves shipping data and code between nodes to even the load on all nodes and improve system throughput. This mechanism can also be used to allocate extra resources to high priority work. This idea has some merit but is not practical for all applications. Often, the best way to implement load balancing is to replicate the software on all possible nodes and trigger execution of each copy by moving data onto that node. In this case load balancing only requires the transfer of data, not code.

27.2 Single Program Models

Many people are currently investigating the feasibility of a single program approach for distributed systems. When a single program spans nodes that do not share a common memory space, communication delays and communication errors between the elements of a program become common events. The programming language and associated run-time system must provide facilities to handle these situations.

The general distributed system issues discussed in Section 27.1 do not lose their importance but become problems that must be solved by the run-time system (as opposed to the application). While this can create problems for the designers of the run-time system, it does relieve the application programmers of a significant burden. This shift in responsibility may be a mixed blessing however; these decisions may affect the capabilities of the final system. Therefore, application programmers may need to retain some control over the resolution of these issues.

Most single program efforts involve designing a distributed processing language such as Argus [LIS87]. These languages generally have a construct that serves as a unit for distribution; for Argus, the Guardian construct. The semantics of this construct facilitate the actual distribution of program elements to different nodes. Ada does not have a construct well suited to this purpose. This lack does not mean that Ada is inadequate for this type of programming, only that the problems encountered will be more difficult.

The first decision in distributing an Ada program is the unit of allocation. Several options have been suggested:

- Tasks.
- Packages.
- Compilation units, and
- Unrestricted.

Choosing either tasks or packages leads to very constrained coding styles. For example, if a task is chosen as the distribution unit, then each task must be associated with a processing node. The only way to have any data or code resident on a node is to associate this data/code with a task on this node. While this design allows communication protocols to be built around the rendezvous mechanism, there are nevertheless serious problems. First, tasks are created with no logical meaning, and second, code and data are forced into tasks, even though tasks are not necessarily the most appropriate Ada construct. Compilation units are a somewhat better choice but still constrain the distribution scheme. Honeywell ([COR84], [KJE87]) has chosen an intriguing method of

allocation. They have developed a distribution language that allows the independent allocation of each part of an Ada program.

Data allocation to a particular node can be done by any of the four methods mentioned above. In the first three cases, data is allocated as part of a large unit that contains both program and data, whereas in the last case, data can be allocated independently. Once data is allocated, two questions arise:

- "How does one reference data allocated to another node?", and
- "Is replicated data supported?"

When a procedure on one node references data on another node, problems can occur. First, the timing behavior of variable references will be unpredictable. More importantly, there is no mechanism to handle either long delays in response or communication errors involving the loss of the read request. Special purpose mechanisms can be built into a distributed run-time system, but they will limit the portability of the resulting software.

The replication of data can have many desirable effects, but the support required for it to meet Ada semantics would be very expensive in terms of processor power and system survivability. It may be necessary to introduce a separate piece of software such as the Distributed Data Management System, discussed in [DDM87], to manage replicated data objects.

Code must also be allocated to nodes. Any of the four general strategies can be used. Code replication raises some interesting issues. There are two types of code in Ada: code that cannot be executed concurrently (the body of a task¹⁹), and code that can be executed concurrently (everything else). There is no reason not to replicate a sub-program, but replicating a task could significantly change the meaning of a program²⁰. If the unit of allocation is a package and the package contains both procedures and tasks, replication may be difficult. If the unit of allocation is a task, the concurrency of associated procedures may be unnecessarily limited. If the unit of allocation is a task (each task and its related procedures are allocated to a particular node) the concurrency of associated procedures may be unnecessarily limited. When several task objects are generated from a single task type, these objects can be on different nodes (and therefore running concurrently). These objects can be allocated to the same node or different nodes, however any single object can reside on one and only one node. The implication is that in order to interact with one of these task objects the code must go through the communication process on the node on which the task object resides. However, proce-

¹⁹Ada task bodies are not reentrant.

²⁰Additional code would be needed to make sure that while the task is running on the first node, it has not also started running on the second node. This check could be very important if the task in question were protecting a shared resource.

dures associated with a task type can be replicated on each node on which a task object of that type is located.

Communication to a remote procedure is similar to referencing a remote variable. Ada does not prevent it, but there is nothing in Ada to define how communications delays or errors should be handled.

Calling an entry on a remote task can cause serious problems if communication errors are encountered. There are several semantic rules about entry calls that can create problems. For example, the time limit on a timed entry call is defined in terms of when the call enters the called task's entry queue, not when it is issued. It is not clear whether a time out could ever be declared due to lost messages or other communication problems. If the call is delayed by the communication system, it is not clear who should time the call out, or when the timeout should occur.

The single program approach has many strengths:

- Ease of application development,
- Reliability,
- Maintainability, and
- Portability.

It also has some important difficulties to overcome. Ada is not the perfect language for this type of approach, especially when timing delays become long and communication faults are encountered. Ada can be adapted to this environment, however, and significant work has been done to make this type of development in Ada a reality. Honeywell, among others, has done significant work in this area.

Section 28

Real-Time Issues in Ada

Real-time systems are computer systems that interact with ongoing real world events. These systems differ from other computer applications in that timeliness is as important as correctness. Historically the production of real-time software has been approached quite differently: while most software is structured according to its functional requirements, real-time software is structured according to its timing requirements. This development method has in turn led to high integration and maintenance costs.

Since the advent of Ada, there has been intense discussion of whether or not real-time systems can be programmed in Ada. The principal aspects of this discussion focus on whether traditional development techniques can map into Ada constructs and on whether Ada implementations can generate efficient code. Ada has the features to support the traditional real-time implementations, as discussed in Section 28.1. Further improvements are needed both in the run-time technology and real-time system development methods to meet the severe memory and time constraints. Recently scientists have been investigating other ways of developing real-time software to reduce cost and increase flexibility.

28.1 Evaluation of Cyclic Executive Approach

Classical real-time development involves the production of a cyclic executive. In this scheme, all processing is assigned specific times in a processing cycle that is executed repeatedly (see Figure 4).

Other traditional real-time development paradigms exist. Their development characteristics are similar to those of a cyclic system, though the specific mechanisms may vary.

28.1.1 Description

A cyclic executive is a mechanism for prescheduling the processing in a system. It provides a single static processing schedule that has been specifically tuned to meet the timing requirements of the application. All processing to be performed is assigned time in a schedule of finite duration. This schedule is repeated at a specified rate, known as the major cycle. The major cycle is broken down into a number (usually a power of two) of minor cycles. Each minor cycle has a processing frame assigned to it. A frame is a list of processing elements to be performed during the associated minor cycle.

There are many variations of cyclic executives, including changing frame assign-

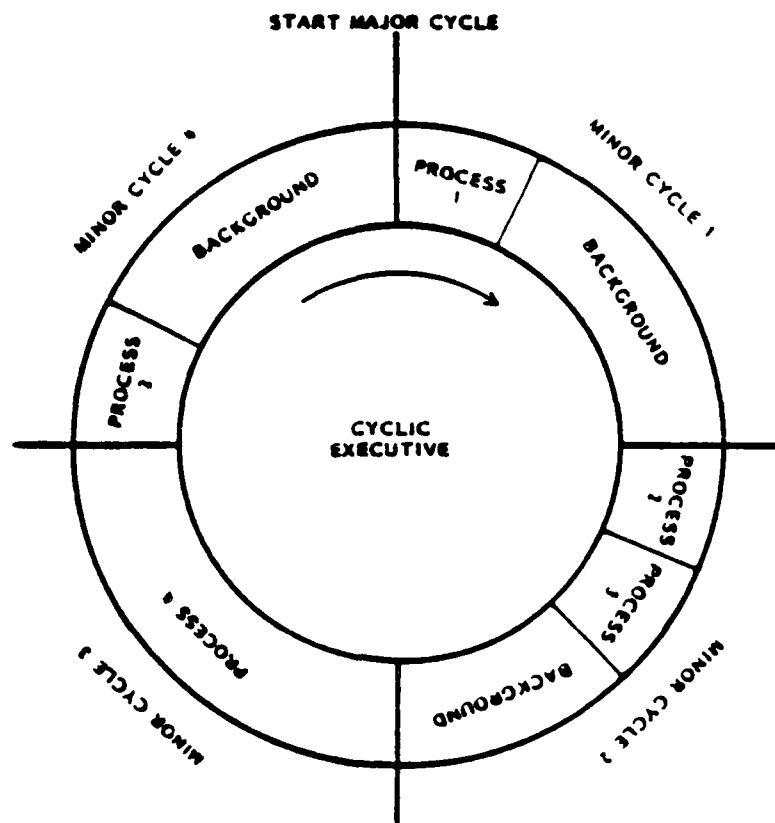


Figure 4: Cyclic Executive Structure

ments during run-time, alternatives for dealing with frame overrun, and handling interrupt and background activities.

Reasons for Choosing this Approach A cyclic executive approach produces code that is efficient and predictable. These are key requirements for most real-time systems. Many real-time systems are based on periodic sampling and control for which the cyclic executive is ideally suited. Cyclic systems are preplanned; every deadline is anticipated and necessary resources are allocated in advance. Efficiency is achieved by making scheduling decisions prior to run-time. Application code can be optimized to take advantage of the inherent synchronization in cyclic systems. (Synchronization of access to shared resources is done prior to run-time.) Timing behavior of a cyclic system is easily analyzed; the time for each frame is the sum of the execution times of its components; the time limit for each frame is a minor cycle. Finally, the behavior of a cyclic system is very deterministic, because the schedule is the same every time. The timing of the system cannot vary much, a highly desirable property for real-time systems.

Ada Implementation Highlights Some varieties of cyclic executives are easily implemented in Ada, others are more difficult. MacLauren [MAC80] and Hood [HOO86] show how a number of cyclic executives can be written in Ada. Implementation problems are encountered when an executive must terminate or suspend overrunning frames. These executives can be written if they are adequately supported by the underlying run-time system. These are excellent candidates for a customized run-time system as discussed in Edition 1 Section 3.3.

28.1.2 Strengths and Weaknesses

The cyclic executive approach is very good for producing real-time systems that work. These systems are unfortunately very expensive, both to produce and especially to maintain. Small changes in software function or computer hardware can mean massive changes to the program. All possible error and overload conditions must be foreseen in advance. Modifying just a few instructions may change the execution time characteristics of the program so that the entire software must be recalibrated. It is not uncommon for seemingly insignificant changes to require multi-million dollar programming efforts (for example, changing to a new processor that is identical to the old processor, except twice as fast).

In summary:

- A cyclic executive allows for precise real-time scheduling.
- Timing analysis is easy to perform and timing errors are detected early when a frame overruns, and
- Cyclic executives are expensive to produce and very expensive to maintain.

28.2 Evaluation of Data Driven Approaches

Recent work has focused on systems that meet real-time requirements without the use of a static scheduling structure like a cyclic executive. This work is exploring several new technologies: data flow machines, functional languages, parallel processing languages, etc. Ada itself reflects an attempt to move in this direction.

28.2.1 Description

All efforts in the data driven methods have one goal: to restore the functional concept to real-time systems. The techniques proposed to engineer this restructuring vary considerably. Functional languages and data flow machines depart from the Von Neumann model of programming by taking the view that data arrival causes an instruction to be fetched and executed. (Von Neumann machines view the instruction as causing data to be fetched and operated upon.) The other approaches are less extreme but center around similar ideas. In all approaches, the flow of data through a system becomes an important factor in system scheduling decisions. It is the presence of data which triggers the scheduler. The analogue in Ada is that tasks which are waiting for data to be passed through a rendezvous are not eligible for execution. Thus the schedule is nondeterministic, introducing an element of unpredictability into the system.

Reasons for Choosing this Approach The data driven type of approach creates real-time software with a functional structure. This software is easier and cheaper to create and maintain. The principal reasons are that the software is inherently more flexible and more adaptable. These properties not only lower the cost of today's systems, but without them, many future systems (such as the SDI or NASA's Space Station) will not be possible. These systems are too large to develop as a single piece; the software must be able to adapt to an evolving environment.

Ada Implementation Highlights Implementations based on the data driven model use Ada tasking extensively. Processing is coded as small Ada tasks that receive data.

process it, and send data out as they finish. If references to shared resources (data and hardware) are allowed at all, they are strictly controlled by monitor tasks. Specialized run-time system optimizations of elements common to this type of system (such as monitor tasks) can greatly improve the speed of this type of system (see Edition 1 Section 3.3).

28.2.2 Strengths and Weaknesses

This type of system does not suffer from the problems of a cyclic system. The data driven approach enjoys the benefits of a highly modular structure. It is robust in the face of changes, is easy to maintain, and easy to adapt to new environments. On the other hand it lacks the efficiency, temporal determinism, predictability, and ease of timing verification and testing that distinguish a cyclic system. Its heavy reliance on Ada tasking incurs the penalty of frequent context switching overhead needed to support the many rendezvous. As the data load on the system increases, the likelihood of the system thrashing and becoming nonresponsive increases.

28.3 Temporal Models

The critical nature of timing requirements for real-time systems has motivated research in the area of temporal models.²¹ By developing sophisticated timing analysis tools, the process of tuning the software to meet its timing constraints can be automated. Current investigations center on hybrid systems which use a data driven development approach and a cyclic run-time approach. The software methodology key lies in the development of an abstract style of data driven programming, without unnecessarily constraining the temporal behavior of a program. The key to the run-time component lies in creating a method of transforming code written in this style into a cyclic executive system. The data driven code provides the flexibility and adaptability, and the transformation allows the final system to take advantage of the predictability and determinism of the cyclic structure.

28.3.1 Processing Models

In order for this approach to work, a programming style must be defined which allows the programmer to specify all temporal properties necessary for program correctness but which does not otherwise constrain the program's timing. This approach has been called the processing model. This model defines a programmer view of a generic computer resource. Specific machine capabilities and specific timing requirements are

²¹ Model of the timing characteristics of a piece of software.

introduced during the transformational step, through which detailed timing behavior is derived. The processing model is similar in many respects to a functional language; however, it is a development abstraction only, not a run-time model.

28.3.2 Transformational Techniques

Several people have explored transformational programming techniques, Cheatham [CHE84], Boyle [B&M84], etc. Transformation into a cyclic executive can build on these techniques combined with the methods used by today's real-time designers to create cyclic systems. Ward [WAR78] discusses a system that uses these techniques to transform very high level specifications of control systems into real-time systems.

A method of tuning software need not be limited to cyclic transformations. While there may always be a class of real-time systems that require cyclic run-time performance, there are other systems that do not require such extreme measures. Many of these systems would benefit from the flexibility of a run-time scheduler²². These systems still require tuning, but not to the same extent. For these systems other tuning transformations can be used. These might include replacing monitor tasks with semaphores, or simplifying groups of tasks using program inversion techniques [RAJ83]. Using these techniques, a system can be tuned until the appropriate level of predictability and efficiency has been reached.

28.4 Run-Time Environment Technology

It is generally recognized that Ada run-time support environments are immature. Shortfalls exist in performance, functionality, and flexibility. Ada run-time systems provide the needed support to pass the Ada Compiler Validation Capability tests but lack optimizations for memory usage and throughput.

The problem is not an Ada language problem but a run-time support problem. The requirements for embedded real-time environments are being analyzed in order to develop an efficient Ada run-time model. For example, by defining in the run-time system a standard set of low-level tasking primitives supporting deterministic scheduling, the programmer gains better control of task scheduling and, therefore, of system timing. An international workshop was conducted in the United Kingdom in May 1987 to discuss technical issues and to recommend solutions, where a consensus could be achieved. The workshop Proceedings [WOR87] are available through Ada Letters. The discussions addressed:

²²A run-time scheduler is that portion of the operating system which determines the order of execution of application code and system software services.

- predictable scheduling,
- tasking efficiency,
- distributed processing, and
- asynchronous exceptions.

A second Workshop is planned for 1988.

The SIGAda Working Group investigating these issues, ARTEWG, has produced four documents²³:

- *A Canonical Model and Taxonomy of Ada Run-time Environments,*
- *The Catalogue of Ada Run-time Implementation Dependencies,*
- *The Catalogue of Interface Features and Option for the Ada Run-time Environment,* and
- *The First Annual Survey of Mission Critical Application Requirements.*

These documents may be obtained by writing to:

Mike Kamrad
Honeywell Systems and Research Center
M/S MN17-2351
3660 Marshall St. NE
Minneapolis, MN 55518
(612) 782-7321
mkamrad@AJPO.SEL.CMU.EDU

On-line information on ARTEWG activities is available in the directory /artnews on the AJPO.SEL.CMU.EDU machine.

²³The research which led to these newly released publications is described in Edition 1, Section 3.3.2.

Section 29

Contractor Evaluation

In evaluating a contractor, certain general qualities should be present. The contractor should:

- have a complete life-cycle oriented plan for management, configuration management, and quality assurance,
- demonstrate a willingness to apply and write reusable components,
- show correctness and usage standards for code,
- practice as well as preach methodologies,
- have good knowledge of software tools and methodologies to promote smooth transitions between phases, and
- have well-trained and supervised personnel.

Evaluating program development plans or on-going work should not be done in a vacuum but should be done in the context of known trends from previous program evaluations. Metrics are being used to evaluate projects, and the resulting information from these can be used to assess contractor development plans and on-going efforts.

29.1 Software Engineering Exercise

The Software Engineering Exercise (SEE) was developed by MITRE to aid in bidder evaluation for the Air Force Electronic Systems Division (ESD) [AMN87]. It is used to assess a contractor's software development approach, give a preview of what will be developed during the contract, and identify potential problems.

The first time it was applied to a source selection was for the Command Center Processing and Display System - Replacement (CCPDS-R) program. The government identified risks that would be associated with the use of Ada, in particular the ability to use Ada effectively at the early stages of software development.

For the CCPDS-R program, each offerer was requested to "provide a prototypical example of the bidder's proposed software development approach." The purpose of the SEE was to allow the government to assess the contractor's approach through its demonstration. The contractor was not expected to prototype an actual system but to prototype the practice of his methodology. The exercise was conducted following

DoD-STD-2167 guidelines. The government evaluated neither all the phases of the life cycle nor the support functions such as quality assurance and configuration management. The evaluation process stressed the contractor's approach to management, and to the requirements and design analyses phases, as reflected in the products required of the participants.

MITRE performed a dry run of the Software Engineering Exercise prior to requesting it of the offerers. The dry run helped the government determine what to request of the offerer, what guidelines to give the offerer, and development of technical evaluation guidelines. Also from the dry run it was reaffirmed that the following three things require enhanced management attention: requirements analysis, Ada tasking, and managing a development team.

The following items were stated conclusions learned from the SEE:

- It was an excellent training mechanism for the government participants.
- It very successfully met its stated objectives,
- It resulted in improved SDPs and more knowledgeable offerer staff.
- It required considerable government preparation and careful evaluation, and
- It was a software engineering exercise, not an Ada exercise.

If SEE were used on a different project, it would need to be modified to reflect the requirements of that particular project. For the CCPDS-R program the SEE was used during source selection, although several ESD programs are considering incorporating SEE as a contract task.

29.2 Ada Decision Matrix

Contractor evaluation involves risk assessment along technical, acquisition and economic lines. A disciplined, objective method of analyzing the risks of using Ada on a program is needed. A probability based approach has been developed by The Aerospace Corporation and documented in [BAK84]. This work, known as the Ada Decision Matrix, has since been automated. The point of contact is:

Dixie B. Baker
The Aerospace Corporation
M5-562
El Segundo, CA 90245
(213) 336-4059

The Ada Decision Matrix involves a project risk worksheet and a risk priority rating worksheet. The project risk potential is essentially a measure of the probability of success, in other words the probability that a given factor is not a risk factor for the project. By weighting the confidence levels in individual criteria with their priority, an overall confidence rating is computed. This final number is *not* an indication of the likelihood of success or failure of a particular project. It is an indication of whether the decision to use Ada will entail low or high risk. This Ada risk must then be evaluated against other, non Ada-related risk factors previously identified for the project. Ideally several evaluators should complete the Decision Matrix in order to remove individual biases for or against the use of Ada.

Each worksheet item is graded on a scale of five confidence levels, ranging from "very low" (a rating of 0.0) to "very high" (a rating of 1.0). Guidance criteria are provided to aid the evaluator in selecting the appropriate rating. Topics addressed include tool availability and quality, staffing, training, life cycle management procedures, and cost. In order to arrive at the probability of success for the technical factors, the evaluator may first want to probe further by asking the kinds of detailed questions found in reports on Ada compiler evaluation, such as KEAS1 and HHMS6.

29.3 Process Evaluation

It is frequently said that the only way to meet the ever increasing demand for software is to increase productivity. Increased productivity is achieved when higher quality software with fewer errors and lower maintenance costs is produced. Judicious use of metrics helps the program manager monitor the status of a project, the productivity of its staff, and the quality of the outputs.

29.3.1 Metrics

The application of productivity measures and quality standards has been the focus of much attention lately, in particular, in the context of contractor evaluation. The Defense Science Board Task Force Report on Military Software [DSB87], for example, strongly recommends both the use of metrics to assess software quality and progress and their routine incorporation into contracts.

The Air Force Electronic Systems Division (ESD) has developed a set of software reporting metrics. This effort was motivated by software acquisition goals as well as the Air Force System Command Product Assurance Initiative. The metrics are designed to measure technical and management aspects of the coding, testing, and operational phases of software development [KEN87]. These metrics include:

- program size (in lines of code),
- staffing,
- software complexity,
- development progress,
- testing progress,
- computer resource,
- program volatility,
- incremental release,
- code change rate, and
- problem reports.

The use of these metrics needs to be tailored according to the objectives of a project. Metrics are statistics and, therefore, they should not be interpreted as absolute measures of progress. They are good trend indicators and can aid the government in monitoring contract health. In all cases, it is important that the government and contractor communicate clearly and understand one another's goals and positions. When metrics are used, both parties must agree on the purpose of the metric, the data to be gathered, and the definition of the metric.

29.3.2 Contractor Capabilities

Prior to contract award, contractor evaluation requires evaluation of the proposed processes to produce the software. The premise is that given a quality development process, at the very least, acceptable software will be written. Assuming that the same processes were used on other projects, it may be possible to evaluate the quality of those products in order to extrapolate whether the organization produces reliable software within cost and budget.

Management control is a cornerstone of the software process. Financial control of the project alone is not sufficient. Program management is required not only to coordinate resources but also to enforce the use of software engineering standards, methods, and tools. Management must address the software issues, including methods, reusability, efficiency, correctness, etc. The management function undertakes planning the software project, and the manager's responsibilities include both creating a realistic schedule and

developing contingency plans. Schedules should not be based on the assumption of perfect software after the first compilation. Test failures, tool inadequacies, and resource mismatches should be accounted for in the planning stages.

The quality goals and acceptance criteria for the project must be established at the beginning of the project, before a contract is even signed. The contractor's management plans should show what methods will be used to achieve which goals and what corrective actions will be taken when a goal is not met. The metrics to be used to validate progress on individual goals should be stated. Moreover, the plan should address how the manager will use the metrics to exercise control over the project, for example, to revise the estimate to complete. In other words, measuring aspects of a project is not sufficient; the use of the metrics should be incorporated into the management plan.

The software methods used should be clearly outlined. The methods selected should be matched to the application area. For example, for extremely complex software, the methodology needs detailed guidance for analyzing the "middle part;" it should not be vague, relying on expertise and magic instead of discipline. The methodology proposed should be suited to the use of Ada as a design and programming language. Specifically, it should address the allocation of software among the different classes of Ada program units, in particular the choice of packages and tasks.

The methodology should describe technical progress metrics in order to track what portion of the analysis, design, testing, etc., is complete. Often such metrics are expressed in terms of the manager's idea of what percent of the activity is complete. Such measures tend to be inaccurate. A more appropriate measure is based on earned value, for example, number of subsystems designed, number of modules on which code reading has been done, or milestones completed²⁴. Milestones should be tailored to the evolutionary nature of Ada development. Software development phases overlap, making it hard to freeze all modules at an identical level of refinement. Critical modules should be designed, coded, and tested first, implying that they may be ready before other subsystems are fully designed.

Review of the draft products and their refinements is needed. This review should be performed by qualified technical personnel; its purpose is to exchange ideas and uncover errors as early as possible. It is likely that several methods will be needed in order to cover the entire software development life cycle. The outputs of one may not necessarily be in the correct input form for the succeeding method; therefore, careful thought is required on how these methods will be integrated. Automation of methods is desirable in order to facilitate tracking and updating of the products. Such automation should include some sort of machine processing of the product in order to verify conformance

²⁴In addition to major program milestones such as preliminary and critical design reviews, the management plan should identify individual mini-milestones within tasks, for which "credit" is taken as they are completed. Examples include monthly status reports, internal publication of white papers, and subsystem start and completion.

to the rules of the method.

Project personnel should be trained in the process and methods to be used on the project. The degree and level of training should be appropriate to the individual's role in the project. For example, the program manager should not take a three week Ada language course, whereas intensive training in a specific design analysis method is appropriate for senior project engineers.

Quality assurance control should be exercised through an independent reporting structure. Quality procedures should be DoD-STD-2167 compliant, addressing the products specific to each phase, including reviews, milestones, code and documents. They should specify acceptance criteria for the individual software-related items and detail checklists to be followed to verify that these standards are being met. There should be procedures in place to evaluate the application of the selected software methodology as well as to exercise management and technical control over subcontractor performance.

The contractor should have a comprehensive configuration management plan. The important criterion here is that configuration management be applied to all phases of the software life cycle. Typically, the source code configurations are carefully managed, but insufficient care is given to the management of the other elements of the product being developed. Documentation at all levels should reflect the software as built; for example, design information and user manuals should be updated when a code change is incorporated.

Appendix A

References

- AFC87] Ada Education and Training Study. Armed Forces Communications and Electronics Association, Volume 1, July 1987.
- AMN87] AdaJUG Meeting Notes, Ada-JOVIAL Users Group, Miamisburg, OH July 13-17, 1987.
- A&W87] Altman, N., and N. Weideman, "Timing Variation in Dual Loop Benchmarks." Technical Report CMU/SEI-87-TR-21, Software Engineering Institute, Pittsburgh, PA, October 1987.
- ALT87] Altman, N. "Factors Causing Unexpected Variations in Ada Benchmarks." Technical Report CMU/SEI-87-TR-22, Software Engineering Institute, Pittsburgh, PA, October 1987.
- A&S87] Amiguet, C., and A. Schiper, "Discrete-Event Simulation in Ada." Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston, MA, December 1987.
- ART87] ARTEWG, "The Challenge of Ada Run-time Environments." Ada Expo Proceedings, Boston, MA, December 1987.
- ADA85] Ausnit, C.N., N.H. Cohen, J.B. Goodenough, and R.S. Eanes, Ada in Practice, Springer-Verlag, New York, 1985.
- BAK84] Baker, D.B., "Ada Decision Matrix," Aerospace Report No. TOR-0084(4453 06)-1, The Aerospace Corporation, March 1984.
- BGA88] Brophy, C.E., S. Godfrey, W.W. Agresti, and V.R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Sixth National Conference on Ada Technology, March 1988.
- BOE87] Boehm, Barry W., "Improving Software Productivity," Computer, September 1987.
- B&M84] Boyle, James M., and Monagur N. Muralidharan, "Program Reusability Through Program Transformation," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September, 1984.
- BRAUN88] Braun, Christine L., "Minimizing Ada Risks Through Benchmarking," Proceedings, The Sixth National Conference on Ada Technology, Arlington, VA, March 14-17, 1988. To be published.
- BRU84] Bruno, Giorgio, "Using Ada for Discrete Event Simulation," Software - Practice and Experience, Vol. 14(7), July 1984.

- [CHE84] Cheatham, Thomas E., Jr., "Reusability Through Program Transformations," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984.
- [CDV86] Clapp, Russell M., Louis Duchesneau, Richard A. Volz, Trevor N. Mudge, and Timothy Schultze, "Toward Real-Time Performance Benchmarks for Ada," Communications of the ACM, Vol. 29, No. 8, pp. 760-778, August 1986.
- [COR84] Cornhill, Dennis, "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," Proceedings of the IEEE Conference on Ada Applications and Environments, 1984.
- [C&W76] Curnow, H.J., and B.A. Wichman, "A Synthetic Benchmark," Computer Journal, p. 43, February 1976.
- [DAU87] Dausmann, Manfred, "Library Structures for Reusable Components," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston, MA, December 1987.
- [DZM87] DiGrazia, Joseph C., J. Ziegler, and R. Mueller, "An Ada Distributed Multiprocessor Executive: From Conceptualization to Implementation," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston, MA December 1987.
- [DDM87] "A Distributed Database Management System for SDI BM/C3 Applications," SofTech Document Number 1142-10, November 25, 1987.
- [DON87] Donohoe, Patrick, "A Survey of Real-Time Performance Benchmarks for the Ada Programming Language," Technical Report, CMU/SEI-87-TR-28, Software Engineering Institute, Pittsburgh, PA, October 1987. To be published.
- [DSB87] Report of the Defense Science Board Task Force on Military Software, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., September 1987.
- [FOF87] Fukuyama, Shunichi, Naoi Okuse, Masato Fujimaru, Seiichi Yamasaki, "Empirical Guidelines to Use Ada Effectively," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston, MA, December 1987.
- [H&S87] Harbaugh, Sam, and Greg Saunders, "GKS/Ada Post Mortem, a Cost Analysis," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.

- [HAS88] Hasekioglu, O., "Queueing Network Modeling and Simulation." Proceedings of the Sixth National Conference on Ada Technology, March 1988.
- [HHM86] Hogan, M.O., E.P. Hauser, and S.P. Menichiello, "The Definition of a Production Quality Ada Compiler," Aerospace Report No. TOR-0086(6902-03)-1. The Aerospace Corporation, El Segundo, CA September 1986.
- [HOO86] Hood, Philip, "Ada and Cyclic Run-time Scheduling," First International Conference on Ada Programming Language Applications, May 1986.
- [KIM88] Kim, ILt L.S., "Advanced Ada Tasking Techniques for Motor Simulation and Control," Proceedings of the Sixth National Conference on Ada Technology, March 1988.
- [KJE87] Kamrad, Mike, Rakesh Jha, Greg Eisenhauer, and Dennis Cornhill, "Distributed Ada," Proceedings of the International Workshop on Real-Time Ada Issues, Ada Letters, Volume VII, Number 6, Fall 1987.
- [KEA84] Kean, E., "Evaluation Criteria for Ada Compilers," RADC, September 1984.
- [KEN87] Kent, R.J., "Software Reporting Metrics," ESD/MITRE Technology Initiatives, AFCEA, Lexington-Concord Chapter, Bedford, MA, June, 1987.
- [LIS87] Liskov, Barbara, Argus Reference Manual, Programming Methodology Group, Memo 54, MIT Laboratory for Computer Science, Cambridge, MA, March 1987.
- [M&G87] Melde, J.E. and P.G. Gage, "Large System Simulation Using Ada," Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston MA, December 1987.
- [MAC80] MacLauren, Lee, "Evolving Toward Ada in Real-Time Systems," SIGPLAN Notices, 1980.
- [N&S87] Nielsen, Kjell W., and Ken Shumate, "Designing Large Real-Time Systems with Ada," Communications of the ACM Vol. 30, No. 8, August 1987.
- [RAJ83] Rajeev, S., "On Applying Ada to Real-Time Systems: The Inversion Technique and Some Examples," Technical Report TP-148, SofTech, Inc., Waltham, MA, March 1983.
- [REI87] Reifer, Donald J., "Ada's Impact: A Quantitative Assessment," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, Boston, MA, December 1987.

- [R&Z83] Rossi, G.F., and R. Zicari, "Programming a Distributed System in Ada," Journal of Pascal and Ada, September/October 1983.
- [SHO87] Shore, R.W., "Discrete-Event Simulation in Ada: Concepts," Ada Letters, Vol. VII, No. 5, September/October 1987.
- [WAR78] Ward, Stephen A., "An Approach to Real-Time Computation," Proceedings of the IEEE Seventh Texas Conference on Computing Systems, October 1978.
- [WEI84] Weicker, Reinhold P., "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, Vol. 27, No. 10, pp. 1013-1030, October 1984.
- [WOR87] Proceedings of the International Workshop on Real-Time Ada Issues, May 1987, Ada Letters, ACM Press, Boston, MA, Fall 1987.

Appendix B

Bibliography

- Ada Information Clearinghouse Newsletter, (AdalC), Vol. V, No. 3, December 1987.
- Ada for Software Managers, Volumes I and II, U.S. Army Communications Electronics Command,
- AFSC/AFLC, AFR 800-11, Supplement 1, September 1987.
- Anthes, Gary H., "Benchmarking," Federal Computer Week, January 11, 1988.
- Auty, D., and Cohen, N., "Establishing an Ada Run-time Benchmarking Capability," Technical Report TP-237, SofTech, Inc., Waltham, MA, March 1987.
- Bachman, Brett, "Design Automation for Ada Development Under DoD-STD-2167 (and Beyond)," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Coles, R.J., J.A. Kasputys, K.L. Lasko, I.F. Saunders, and H.P. Schultz, Software Reporting Metrics, Revision 2, ESD-TR-85-145, Electronic Systems Division, Hanscom AFB, MA, November 1985.
- Cohen, Norman H., Ada as a Second Language, McGraw-Hill, New York, 1986.
- Conceptual Design of a Distributed Data Management System, Software Requirements Document, Document Number 1142-5.1, SofTech, Inc., Waltham, MA, September 1987.
- DoD-STD-2167 Defense System Software Development, June 1985.
- DoD Directive 3405.1, Computer Programming Language Policy, April 2, 1987.
- DoD Directive 3405.2, Use of Ada in Weapon Systems, March 30, 1987.
- Foreman, John and Goodenough, John, Ada Adoption Handbook: A Program Manager's Guide, CMU-SEI-87-TR-9, Software Engineering Institute, Pittsburgh, PA, May 1987.
- Franel, F., "Pioneering Mission-Critical Ada Software," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- HQAF/SCT, AFR 700-9, March 1985.

- Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M., and Sherman, M., Studies in Ada Style, Second Edition, Springer-Verlag, New York, 1983.
- Kernighan, B.J., and Plauger, P.J., The Elements of Programming Style, Second Edition, McGraw-Hill, 1978.
- Lucas, L., and Dent, D., "Real-Time Ada Demonstration," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Maxted, A., and Rowe, J.C., "An Ada Graphical Tool to Support Software Development," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Melde, J.E., and Gage, P.G., "Large System Simulation Using Ada," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Moreton, T., "Partitioned Ada Libraries as a Basis for Variant Control," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Murray, L.E., "A Life-Cycle Oriented Ada Design Language," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- SAF/AQX, AFR 800-14, September 1986.
- Schacht, E.N., "Ada Programming Techniques, Research, and Experiences on a Fast Control Loop System," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Schefstrom, D., "The System-Oriented Editor - A Tool for Managing Large Software Systems," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Schultz, W.L., and Chandna, A., "An Ada Based Approach to Factory Scale MAP Network Simulation," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Seriin, O., "MIPS, Dhrystones, and Other Tales," Datamation, June 1, 1986.
- Ternes, D.H., "Developmental Software Configuration and Integration in a Large Ada Project," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.
- Walters, M.D., "Expert Systems Development in LISP and Ada," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.

- Watts, H. and D. Kitson, Preliminary Report on Conducting SEI-Assisted Assessments of Software Engineering Capability, CMU/SEI-87-TR-16, Software Engineering Institute, Pittsburgh, PA, July 1987.
- Waxman, Martin, "Ada, Real Time and the Run-time Environment," AdaData, Vol. 5, No. 9, pp. 12-14, September 1987.
- Weiderman, N., Haberman, N., "Evaluation of Ada Environments," Technical Report CMU/SEI-87-TR-1, Software Engineering Institute, Pittsburgh, PA, March 1987.
- Weiderman, N., "Methodology for the Evaluation of Ada Environments," Technical Report SEI-86-MR-2, Software Engineering Institute, Pittsburgh, PA, March 1987.
- Williams, Charles B., "Use of the Rational(R) R1000(R) Ada Development Environment for an IBM(R) Based Command and Control System," Proceedings, Using Ada: ACM SIGAda International Conference, Ada Letters, ACM Press, December 1987.

Appendix C

Points of Contact for Ada Information

(In presentation order)

PIWG	Jon S. Squire Westinghouse Electric Corporation P.O. Box 746 M/S 1615 Baltimore, MD 21203 (301)765-3748 benchmark@ajpo.sei.cmu.edu
Production Quality Ada Compiler Report Report No. TOR-0086(6902-03)-1	Aerospace Library Reports Circulation (M1-199) P.O. Box 92957 Los Angeles, CA 90009
Application of Production Quality Ada Compiler Report	Lt Kurt Maschoff Space Division SD ALR P.O. Box 92960 Los Angeles, CA 90009 (213) 643-1279 (AV) 833-1279
Real-Time Run-Time Environment Studies	Rich Kayfes Aerospace Corporation (M1/165) P.O. Box 92957 Los Angeles, CA 90009 (213) 336-6092
CAUWG	Dave Dikel Addamax Corporation 7799 Leesburg Pike, Suite 900, Tysons Corner, VA 22043 (703)847-6755

Armonics Benchmarks

DACS
RADC/COED
Griffiss AFB, NY 13441-5700
Attn: Document/Data Set Ordering
(315) 336-0937

SDI Simulation

The MITRE Corporation
Burlington Rd
Bedford, MA 01730
(617) 271-4501

Lessons Learned Briefing

ASD/YWB
Wright Patterson AFB, OH 45433
(513) 255-7177

DoD Ada Programs
Database

Ada Information Clearinghouse
3D139 (1211 Fern St., C-107)
The Pentagon
Washington D.C., 20301-3081
Attn: Ada Usage

ARTEWG Activities
and Documents

Mike Kamrad
Honeywell Systems and Research Center
M, S MN17-2351
3660 Marshall St. NE
Minneapolis, MN 55518
(612) 782-7321
mkamrad@AJPO.SEL.CMU.EDU

Ada Decision Matrix

Dixie B. Baker
The Aerospace Corporation
M5-562
El Segundo, CA 90245
(213) 336-4059